Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

**УПРАВЛЕНИЕ СЕРВЕРАМИ СООБЩЕНИЙ  
ПРИМЕНЕНИЕ ОТЛОЖЕННЫХ ВЫЧИСЛЕНИЙ  
ИНТЕГРАЦИЯ ПРОГРАММНЫХ СИСТЕМ ДРУГ С ДРУГОМ**

Студент: Головенко Анатолий Валерьевич

Группа: М8О–212Б–22

Вариант: 2

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2023.

**Постановка задачи**

Цель работы

Целью является приобретение практических навыков в:

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

Задание

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить свою работоспособность.

Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

Создание нового вычислительного узла

Формат команды: create id [parent]

id – целочисленный идентификатор нового вычислительного узла

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

Формат вывода:

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

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

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

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

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

> create 10 5

Ok: 3128

Примечания: создание нового управляющего узла осуществляется пользователем программы при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы.

Исполнение команды на вычислительном узле

Формат команды: exec id [params]

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

Формат вывода:

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

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

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

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

Пример:

Можно найти в описании конкретной команды, определенной вариантом задания.

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

**Общие сведения о программе**

Исходный код лежит в 11 файлах, еще один - СMakeList:

1. main.cpp - файл, в котором происходит запуск приложения, с помощью вызова метода run у класса SpringBootApplication
2. server.cpp- код серверов(вычислительных узлов).
3. MessageData.h - класс, определяющий содержание сообщения.
4. ZmqUtils.h - класс со статическими утилитными методами.
5. MessageTypes.h - enum, содержащий типы сообщений.
6. MessageBuilder.h - класс, собирающий сообщения по требованию. В нем также содержатся средства сериализации-десериализации.
7. Message.h - класс сообщения.
8. AbstractNode.h - абстрактный узел.
9. SpringBootApplication.h - клиент(управляющий узел). Наследуется от абстракт- ного.
10. ServerNode.h - сервер(вычислительный узел). Наследуется от абстрактного.
11. ChildNodeInfo.h - класс, содержащий представление дочернего узла при хранении в мапе потомков данного.

Программа собирается с помощью CmakeLists.txt. Нужно получитьдва исполняемых файла для клиента и серверов.

**Общий метод и алгоритм решения**.

Имеем два исполняемых файла, но непосредственно работаем только с одним. Другой используется внутри вызова execl для создания дочернего вычислительного узла. Нам нужно организовать асинхронность. Для этого запускаем в клиенте несколько пото- ков конвеерообразно. Первый работает с пользователем, принимает от него команды и через push-pull сокет передает в поток, отвечающий за отправку сообщения и полу- чение резельтата. И если данный поток может надолго заснуть в ожидании ответа от вычислительного узла, первый потенциально никогда не заблокируется надолго и смо- жет продолжать работать с пользователем, заполняя очередь для следующего потока. Кроме этого есть поток, отвечающий за вывод результатов запросов и поток, отвеча- ющий за регистрацию свежесозданных узлов. В дереве общего вида мы не знаем, где расположена нода, на которую идет запрос и как её найти, в отличие от бинарного. Так что каждый узел, начиная с управляющего, при необходимости отправить сообщение проходит по всем дочерним узлам, подключается и отправляет сообщение. Далее узлы выполняют запрос, если он адресован им, или пробрасывают его дальше по дереву. Так запрос волной пройдет по всем узлам и точно найдет тот, которому он адресован, если он, конечно, доступен. Heartbit запускается по команде в отдельном потоке. В идеале узлы должны именно сообщать о том, что они живы. В моем же случае всё межпро- цессное взаимодействие основано на req-rep сокетах, так что узел нужно спросить, жив ли он. Если узел не отвечает указанное время - считаем его мертвым.

**Основные файлы программы**

**AbstractNode.h**

#pragma once

#include <map>

#include <unistd.h>

#include <zmq.h>

#include "ChildNodeInfo.h"

#include <mutex>

#include <utility>

#include "ZmqUtils.h"

using std::to\_string;

class AbstractNode

{

public:

explicit AbstractNode(int id) : Id(id), Port(ZmqUtils::PORT\_TO\_BIND\_FROM),

outerNodes(), context(1), Receiver(context, zmq::socket\_type::rep)

{

occupyPort();

}

AbstractNode() {}

pid\_t addChild(int id, int registerPort)

{

pid\_t child = fork();

if (child == 0)

{

execl("server", to\_string(id).c\_str(), to\_string(Id).c\_str(),

to\_string(Port).c\_str(), to\_string(registerPort).c\_str(), NULL);

}

return child;

}

protected:

int Id;

int Port;

std::map<int, ChildNodeInfo> outerNodes;

zmq::context\_t context;

zmq::socket\_t Receiver;

void occupyPort()

{

Port = ZmqUtils::occupyPort(Receiver);

}

};

**ChildNodeInfo.h**

#pragma once

#include <thread>

#include "MessageTypes.h"

class ChildNodeInfo

{

public:

ChildNodeInfo(pid\_t pid, int messageRecieverPort, int childRegisterPort) : Pid(pid), ReceiverPort(messageRecieverPort),

RegisterPort(childRegisterPort)

{

}

pid\_t Pid;

int RegisterPort;

int ReceiverPort;

};

**CMakeLists.txt**

cmake\_minimum\_required(VERSION 3.20)

project(lab6)

set(CMAKE\_CXX\_STANDARD 17)

find\_package(cppzmq)

add\_executable(lab6 main.cpp AbstractNode.h ChildNodeInfo.h ZmqUtils.h Message.h MessageTypes.h MessageBuilder.h MessageData.h SpringBootApplication.h)

target\_link\_libraries(lab6 cppzmq)

target\_link\_libraries(lab6 pthread)

add\_executable(server AbstractNode.h ChildNodeInfo.h ZmqUtils.h Message.h MessageTypes.h MessageBuilder.h MessageData.h SpringBootApplication.h server.cpp)

target\_link\_libraries(server cppzmq)

target\_link\_libraries(server pthread)

**main.cpp**

#include <iostream>

#include "ServerNode.h"

#include <sys/wait.h>

#include "SpringBootApplication.h"

void child(int sig) {

pid\_t pid;

pid = wait(nullptr);

}

int main() {

signal(SIGCHLD, child);

SpringBootApplication app;

app.run();

}

**Message.h**

#pragma once

#include <zmq.hpp>

#include <chrono>

#include <thread>

#include "MessageTypes.h"

class Message

{

public:

Message(MessageTypes messageType, int senderId, int receiverId, int \_sizeOfBody, void \*\_body) : messageType(

messageType),

senderId(senderId), recieverId(receiverId), sizeOfBody(\_sizeOfBody), body(nullptr)

{

if (sizeOfBody > 0)

{

body = malloc(sizeOfBody);

memcpy(body, \_body, \_sizeOfBody);

}

}

Message() : Message(MessageTypes::EMPTY, -1, -1, 0, nullptr)

{

}

Message(MessageTypes messageType, int senderId, int receiverId) : Message(messageType, senderId, receiverId, 0, nullptr)

{

}

~Message()

{

if (body != nullptr)

{

free(body);

body = nullptr;

}

}

Message(Message &&message) noexcept

{

messageType = message.messageType;

senderId = message.senderId;

recieverId = message.recieverId;

sizeOfBody = message.sizeOfBody;

body = message.body;

message.body = nullptr;

}

Message &operator=(Message &&message) noexcept

{

if (body != nullptr)

{

free(body);

body = nullptr;

}

messageType = message.messageType;

senderId = message.senderId;

recieverId = message.recieverId;

sizeOfBody = message.sizeOfBody;

body = message.body;

message.body = nullptr;

return \*this;

}

Message(MessageTypes messageType, int senderId, int receiverId, const char \*str) : Message(messageType, senderId, receiverId)

{

sizeOfBody = strlen(str) + 1;

if (sizeOfBody > 0)

{

body = operator new(sizeOfBody);

memcpy(body, (void \*)str, sizeOfBody);

}

}

void sendMessage(zmq::socket\_t &socket)

{

int header[] = {senderId, recieverId, sizeOfBody};

zmq::message\_t typeOfMessage(&messageType, sizeof(messageType));

zmq::message\_t messageHeader(&header, sizeof(header));

zmq::message\_t messageBody(body, sizeOfBody);

// send compound message using SNDMORE flag in every message part except the last one

socket.send(typeOfMessage, zmq::send\_flags::sndmore);

socket.send(messageHeader, zmq::send\_flags::sndmore);

socket.send(messageBody, zmq::send\_flags::none);

}

void receiveMessage(zmq::socket\_t &socket)

{

zmq::message\_t receivedType;

zmq::message\_t receivedHeader;

zmq::message\_t receivedBody;

for (bool t = false; !t; t = getMessage(socket, receivedType, receivedHeader, receivedBody))

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

}

messageType = \*(MessageTypes \*)receivedType.data();

senderId = ((int \*)receivedHeader.data())[0];

recieverId = ((int \*)receivedHeader.data())[1];

sizeOfBody = ((int \*)receivedHeader.data())[2];

operator delete(body);

body = nullptr;

if (sizeOfBody > 0)

{

body = operator new(sizeOfBody);

memcpy(body, receivedBody.data(), sizeOfBody);

}

}

bool receiveMessage(zmq::socket\_t &socket, std::chrono::milliseconds time)

{

zmq::message\_t receivedType;

zmq::message\_t receivedHeader;

zmq::message\_t receivedBody;

unsigned long long passed = 0;

bool t = false;

std::chrono::time\_point<std::chrono::system\_clock> start = std::chrono::system\_clock::now();

for (; !t && passed < time.count(); t = getMessage(socket, receivedType, receivedHeader, receivedBody))

{

passed = std::chrono::duration\_cast<std::chrono::milliseconds>(std::chrono::system\_clock::now() - start).count();

std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

}

if (!t)

return false;

messageType = \*(MessageTypes \*)receivedType.data();

senderId = ((int \*)receivedHeader.data())[0];

recieverId = ((int \*)receivedHeader.data())[1];

sizeOfBody = ((int \*)receivedHeader.data())[2];

operator delete(body);

body = nullptr;

if (sizeOfBody > 0)

{

body = operator new(sizeOfBody);

memcpy(body, receivedBody.data(), sizeOfBody);

}

return true;

}

void update(int senderID, int receiverId)

{

senderId = senderID;

recieverId = receiverId;

}

MessageTypes messageType;

int senderId;

int recieverId;

int sizeOfBody;

void \*body;

private:

bool getMessage(zmq::socket\_t &socket, zmq::message\_t &typeOfMessage,

zmq::message\_t &header, zmq::message\_t &bodyOfMessage)

{

return (socket.recv(typeOfMessage, zmq::recv\_flags::dontwait) &&

socket.recv(header, zmq::recv\_flags::dontwait) &&

socket.recv(bodyOfMessage, zmq::recv\_flags::dontwait));

}

};

**MessageBuilder.h**

#pragma once

#include <zmq.hpp>

#include <chrono>

#include <thread>

#include "MessageTypes.h"

class Message

{

public:

Message(MessageTypes messageType, int senderId, int receiverId, int \_sizeOfBody, void \*\_body) : messageType(

messageType),

senderId(senderId), recieverId(receiverId), sizeOfBody(\_sizeOfBody), body(nullptr)

{

if (sizeOfBody > 0)

{

body = malloc(sizeOfBody);

memcpy(body, \_body, \_sizeOfBody);

}

}

Message() : Message(MessageTypes::EMPTY, -1, -1, 0, nullptr)

{

}

Message(MessageTypes messageType, int senderId, int receiverId) : Message(messageType, senderId, receiverId, 0, nullptr)

{

}

~Message()

{

if (body != nullptr)

{

free(body);

body = nullptr;

}

}

Message(Message &&message) noexcept

{

messageType = message.messageType;

senderId = message.senderId;

recieverId = message.recieverId;

sizeOfBody = message.sizeOfBody;

body = message.body;

message.body = nullptr;

}

Message &operator=(Message &&message) noexcept

{

if (body != nullptr)

{

free(body);

body = nullptr;

}

messageType = message.messageType;

senderId = message.senderId;

recieverId = message.recieverId;

sizeOfBody = message.sizeOfBody;

body = message.body;

message.body = nullptr;

return \*this;

}

Message(MessageTypes messageType, int senderId, int receiverId, const char \*str) : Message(messageType, senderId, receiverId)

{

sizeOfBody = strlen(str) + 1;

if (sizeOfBody > 0)

{

body = operator new(sizeOfBody);

memcpy(body, (void \*)str, sizeOfBody);

}

}

void sendMessage(zmq::socket\_t &socket)

{

int header[] = {senderId, recieverId, sizeOfBody};

zmq::message\_t typeOfMessage(&messageType, sizeof(messageType));

zmq::message\_t messageHeader(&header, sizeof(header));

zmq::message\_t messageBody(body, sizeOfBody);

// send compound message using SNDMORE flag in every message part except the last one

socket.send(typeOfMessage, zmq::send\_flags::sndmore);

socket.send(messageHeader, zmq::send\_flags::sndmore);

socket.send(messageBody, zmq::send\_flags::none);

}

void receiveMessage(zmq::socket\_t &socket)

{

zmq::message\_t receivedType;

zmq::message\_t receivedHeader;

zmq::message\_t receivedBody;

for (bool t = false; !t; t = getMessage(socket, receivedType, receivedHeader, receivedBody))

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

}

messageType = \*(MessageTypes \*)receivedType.data();

senderId = ((int \*)receivedHeader.data())[0];

recieverId = ((int \*)receivedHeader.data())[1];

sizeOfBody = ((int \*)receivedHeader.data())[2];

operator delete(body);

body = nullptr;

if (sizeOfBody > 0)

{

body = operator new(sizeOfBody);

memcpy(body, receivedBody.data(), sizeOfBody);

}

}

bool receiveMessage(zmq::socket\_t &socket, std::chrono::milliseconds time)

{

zmq::message\_t receivedType;

zmq::message\_t receivedHeader;

zmq::message\_t receivedBody;

unsigned long long passed = 0;

bool t = false;

std::chrono::time\_point<std::chrono::system\_clock> start = std::chrono::system\_clock::now();

for (; !t && passed < time.count(); t = getMessage(socket, receivedType, receivedHeader, receivedBody))

{

passed = std::chrono::duration\_cast<std::chrono::milliseconds>(std::chrono::system\_clock::now() - start).count();

std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

}

if (!t)

return false;

messageType = \*(MessageTypes \*)receivedType.data();

senderId = ((int \*)receivedHeader.data())[0];

recieverId = ((int \*)receivedHeader.data())[1];

sizeOfBody = ((int \*)receivedHeader.data())[2];

operator delete(body);

body = nullptr;

if (sizeOfBody > 0)

{

body = operator new(sizeOfBody);

memcpy(body, receivedBody.data(), sizeOfBody);

}

return true;

}

void update(int senderID, int receiverId)

{

senderId = senderID;

recieverId = receiverId;

}

MessageTypes messageType;

int senderId;

int recieverId;

int sizeOfBody;

void \*body;

private:

bool getMessage(zmq::socket\_t &socket, zmq::message\_t &typeOfMessage,

zmq::message\_t &header, zmq::message\_t &bodyOfMessage)

{

return (socket.recv(typeOfMessage, zmq::recv\_flags::dontwait) &&

socket.recv(header, zmq::recv\_flags::dontwait) &&

socket.recv(bodyOfMessage, zmq::recv\_flags::dontwait));

}

};

**MessageData.h**

#pragma once

#include <vector>

#include <string>

class MessageData

{

public:

int id;

int len1;

int len2;

std::vector<std::string> data;

};

**MessageTypes.h**

#pragma once

enum class MessageTypes : int

{

CREATE\_REQUEST,

CREATE\_RESULT,

CREATE\_FAIL,

EXEC\_REQUEST,

EXEC\_RESULT,

EXEC\_FAIL,

HEARTBIT\_REQUEST,

HEARTBIT\_RESULT,

HEARTBIT\_FAIL,

RELATE\_RESULT,

RELATE\_REQUEST,

QUIT,

EMPTY,

TEST,

};

**Server.cpp**

#include <zmq.hpp>

#include <unistd.h>

#include <iostream>

#include <signal.h>

#include <sys/wait.h>

#include "SpringBootApplication.h"

#include "ServerNode.h"

void child(int sig)

{

pid\_t pid;

pid = wait(nullptr);

}

int main(int argc, char \*\*argv)

{

signal(SIGCHLD, child);

if (argc != 4)

{

return 0;

}

ServerNode serverNode(atoi(argv[0]), atoi(argv[1]), atoi(argv[2]), atoi(argv[3]));

std::cout << "server created" << std::endl;

serverNode.run();

return 0;

}

**ServerNode.h**

#pragma once

#include "SpringBootApplication.h"

#include "AbstractNode.h"

#include "Message.h"

#include <chrono>

#include "MessageData.h"

#include "MessageBuilder.h"

class ServerNode : public AbstractNode

{

public:

ServerNode(int id, int \_parentId, int \_parentPort, int \_registerPort) : AbstractNode(id), parentId(\_parentId), parentPort(\_parentPort), parentRegPort(\_registerPort),

registerSocket(context, zmq::socket\_type::rep)

{

occupyPort();

registerPort = ZmqUtils::occupyPort(registerSocket);

selfRelation(parentRegPort);

}

void run()

{

std::cout << "SERVER CREATED" << std::endl;

// std::cout << "port " << Port << " parent port " << parentPort << std::endl;

std::thread([this]()

{ registrator(); })

.detach();

bool quit = false;

bool ping = false;

while (true)

{

Message messageOut;

Message messageIn;

// std::cout << "before recieving" << std::endl;

if (!messageIn.receiveMessage(Receiver, std::chrono::milliseconds(100)))

{

continue;

}

// std::cout << "after recevinge" << std::endl;

int time;

std::thread th;

switch (messageIn.messageType)

{

case MessageTypes::CREATE\_REQUEST:

messageOut = createProcessor(messageIn);

messageOut.sendMessage(Receiver);

break;

case MessageTypes::EXEC\_REQUEST:

messageOut = execProcessor(messageIn);

messageOut.sendMessage(Receiver);

break;

case MessageTypes::HEARTBIT\_REQUEST:

// std::cout << "Heartbit request arrived at server " << Id << std::endl;

time = \*(int \*)messageIn.body;

processPingRequest(time);

if (!ping)

{

ping = true;

th = std::thread([this, &time, &quit]()

{ bitGenerator(\*this, time, quit); });

th.detach();

}

messageIn.sendMessage(Receiver);

break;

case MessageTypes::QUIT:

messageOut = quitProcessor(messageIn);

messageOut.sendMessage(Receiver);

quit = true;

exit(1);

default:

break;

}

}

}

private:

int parentId;

int parentPort;

int parentRegPort;

zmq::socket\_t registerSocket;

int registerPort;

std::mutex creationalBlock;

std::condition\_variable creationalCondition;

bool created;

static void bitGenerator(ServerNode &that, int &time, bool &exit)

{

while (!exit)

{

for (auto server : that.outerNodes)

{

zmq::socket\_t sender(that.context, zmq::socket\_type::req);

sender.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

Message pingRequest = MessageBuilder::buildPingRequest(time, server.first);

try

{

pingRequest.sendMessage(sender);

if (!pingRequest.receiveMessage(sender, std::chrono::milliseconds(time \* 4)))

{

std::cout << "node " << server.first << " and all her children dead" << std::endl;

// that.outerNodes.erase(server.first);

}

}

catch (const zmq::error\_t &error)

{

continue;

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(time));

}

}

}

void processPingRequest(int time)

{

for (auto server : outerNodes)

{

Message pingRequest = MessageBuilder::buildPingRequest(time, Id);

zmq::socket\_t socket(context, zmq::socket\_type::req);

socket.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

pingRequest.sendMessage(socket);

pingRequest.receiveMessage(socket, std::chrono::milliseconds(1000));

// std::cout << "message sent to node " << server.first << std::endl;

socket.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

}

}

void registrator()

{

Message messageIn;

std::cout << "from registrator " << getpid() << std::endl;

Message messageOut;

while (true)

{

messageIn.receiveMessage(registerSocket);

std::cout << "register message got" << std::endl;

if (messageIn.messageType == MessageTypes::QUIT)

{

messageIn.sendMessage(registerSocket);

break;

}

messageOut = registerProcessor(messageIn);

ready();

messageOut.sendMessage(registerSocket);

}

}

void ready()

{

std::unique\_lock<std::mutex> lock(creationalBlock);

created = true;

creationalCondition.notify\_one();

}

void \*createRelation(pid\_t pid, int receiverPort, int registerPort)

{

auto mem = new char[sizeof(pid) + 2 \* sizeof(int)];

auto memPid = (pid\_t \*)(mem);

auto memReceiverPort = (int \*)(mem + sizeof(pid\_t));

auto memRegisterPort = (int \*)(mem + sizeof(pid\_t) + sizeof(int));

\*memPid = pid;

\*memReceiverPort = receiverPort;

\*memRegisterPort = registerPort;

return (void \*)mem;

}

void selfRelation(int \_registerPort)

{

zmq::socket\_t sender(context, zmq::socket\_type::req);

sender.connect(ZmqUtils::getOutputAddress(\_registerPort));

void \*relationBody = createRelation(getpid(), Port, registerPort);

Message message(MessageTypes::RELATE\_REQUEST, Id, parentId, 2 \* sizeof(int) + sizeof(pid\_t), relationBody);

message.sendMessage(sender);

message.receiveMessage(sender);

sender.disconnect(ZmqUtils::getOutputAddress(\_registerPort));

delete[] (char \*)relationBody;

relationBody = nullptr;

}

Message registerProcessor(Message &message)

{

auto messageBody = (char \*)message.body;

pid\_t pid = \*(pid\_t \*)(messageBody);

int receiverPort = \*(int \*)(messageBody + sizeof(pid\_t));

int registerPort = \*(int \*)(messageBody + sizeof(pid\_t) + sizeof(int));

outerNodes.emplace(message.senderId, ChildNodeInfo(pid, receiverPort, registerPort));

return Message(MessageTypes::RELATE\_RESULT, Id, message.senderId);

}

Message createProcessor(Message &message)

{

int id = ((int \*)message.body)[0];

int parId = ((int \*)message.body)[1];

int sndId = message.senderId;

if (parId == Id)

{

pid\_t pid = addChild(id, registerPort);

std::unique\_lock<std::mutex> lock(creationalBlock);

created = false;

while (!created)

{

creationalCondition.wait(lock);

}

return {MessageTypes::CREATE\_RESULT, Id, parentId, sizeof(pid), &pid};

}

else if (outerNodes.empty())

{

return {MessageTypes::CREATE\_FAIL, Id, parentId};

}

int pid = -1;

for (auto server : outerNodes)

{

zmq::socket\_t sender(context, zmq::socket\_type::req);

sender.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

int data[] = {id, parId};

Message request(MessageTypes::CREATE\_REQUEST, Id, server.first, sizeof(data), (void \*)data);

Message result;

request.sendMessage(sender);

// result.receiveMessage(sender);

if (!result.receiveMessage(sender, std::chrono::milliseconds(2000)))

{

sender.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

continue;

}

if (result.messageType == MessageTypes::CREATE\_RESULT)

{

pid = \*(int \*)result.body;

sender.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

break;

}

sender.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

}

if (pid == -1)

{

return {MessageTypes::CREATE\_FAIL, Id, sndId, sizeof(pid), &pid};

}

return {MessageTypes::CREATE\_RESULT, Id, sndId, sizeof(pid), &pid};

}

Message execProcessor(Message &message)

{

MessageData data = MessageBuilder::deserialize(message.body);

std::string text = data.data[0];

std::string pattern = data.data[1];

// std::cout << "EXEC INFO: Id " << Id << "target id " << data.id << std::endl;

if (data.id == Id)

{

text = text.substr(0, data.len1);

pattern = pattern.substr(0, data.len2);

size\_t found = text.find(pattern, 0);

std::string answer;

if (found != std::string::npos)

{

while (found != std::string::npos)

{

answer += std::to\_string(found);

answer += " ";

found = text.find(pattern, found + 1);

}

std::cout << answer << std::endl;

}

else

{

std::cout << "NOT FOUND!" << std::endl;

}

std::vector<std::string> newData(2);

newData[0] = answer;

newData[1] = "";

void \*newBody = MessageBuilder::serialize(data.id, newData);

int newSize = MessageBuilder::getSize(newData);

std::cout << "RECEVIEd!! " << text << " " << pattern << std::endl;

return {MessageTypes::EXEC\_RESULT, Id, parentId, newSize, newBody};

}

for (auto server : this->outerNodes)

{

zmq::socket\_t requester(context, zmq::socket\_type::req);

requester.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

message.update(Id, server.first);

message.sendMessage(requester);

Message result;

// result.receiveMessage(requester);

if (!result.receiveMessage(requester, std::chrono::milliseconds(2000)))

{

requester.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

continue;

}

if (result.messageType == MessageTypes::EXEC\_RESULT)

{

result.update(Id, parentId);

requester.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

return result;

}

requester.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

}

return {MessageTypes::EXEC\_FAIL, Id, parentId, "node is unaviable"};

}

Message quitProcessor(Message &message)

{

Message messageQuit(MessageTypes::QUIT, Id, Id);

for (auto server : outerNodes)

{

zmq::socket\_t request(context, zmq::socket\_type::req);

request.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

messageQuit.sendMessage(request);

Message messageResult;

// messageResult.receiveMessage(request);

if (!messageResult.receiveMessage(request, std::chrono::milliseconds(2000)))

{

request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

continue;

}

request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

}

zmq::socket\_t request(context, zmq::socket\_type::req);

request.connect(ZmqUtils::getOutputAddress(registerPort));

messageQuit.sendMessage(request);

// messageQuit.receiveMessage(request);

messageQuit.receiveMessage(request, std::chrono::milliseconds(2000));

request.disconnect(ZmqUtils::getOutputAddress(registerPort));

return messageQuit;

}

};

**SpringBootApplication.h**

#pragma once

#include <set>

#include <mutex>

#include <condition\_variable>

#include "ServerNode.h"

#include "AbstractNode.h"

#include "Message.h"

#include "ZmqUtils.h"

#include "MessageBuilder.h"

class SpringBootApplication : public AbstractNode

{

public:

SpringBootApplication() : AbstractNode(-1), created(true)

{

serversId.insert(-1);

}

void run()

{

std::thread([this]()

{ inputProcessing(); })

.detach();

std::thread([this]()

{ messageProcessing(); })

.detach();

std::thread([this]()

{ registrator(); })

.detach();

messageOutput();

}

private:

std::set<int> serversId;

std::mutex creationLock;

std::condition\_variable creationalCondition;

bool created;

void inputProcessing()

{

std::string command;

zmq::socket\_t toMessageProcessor(context, zmq::socket\_type::push);

toMessageProcessor.connect(ZmqUtils::MESSAGE\_PROCESSOR\_URL);

while (true)

{

std::cin >> command;

if (command == "create")

{

Message message = MessageBuilder::buildCreateMessage(Id);

message.sendMessage(toMessageProcessor);

}

if (command == "exit")

{

Message message = MessageBuilder::buildExitMessage(Id);

message.sendMessage(toMessageProcessor);

break;

}

if (command == "exec")

{

Message message = MessageBuilder::buildExecMessage();

message.sendMessage(toMessageProcessor);

}

if (command == "hb")

{

int time;

std::cin >> time;

Message message = MessageBuilder::buildPingRequest(time, -1);

message.sendMessage(toMessageProcessor);

}

else

{

std::cin.clear();

}

}

}

void messageProcessing()

{

zmq::socket\_t fromInputProcessor(context, zmq::socket\_type::pull);

zmq::socket\_t toOutput(context, zmq::socket\_type::push);

fromInputProcessor.bind(ZmqUtils::MESSAGE\_PROCESSOR\_URL);

toOutput.connect(ZmqUtils::MESSAGE\_SENDER\_URL);

MessageData data;

std::thread pl;

Message toSend;

bool quit = false;

while (!quit)

{

Message receivedMessage;

receivedMessage.receiveMessage(fromInputProcessor);

int time;

std::thread th;

switch (receivedMessage.messageType)

{

case MessageTypes::CREATE\_REQUEST:

toSend = processCreateMessage(std::move(receivedMessage));

toSend.sendMessage(toOutput);

break;

case MessageTypes::EXEC\_REQUEST:

toSend = processExecMessage(std::move(receivedMessage));

toSend.sendMessage(toOutput);

break;

case MessageTypes::HEARTBIT\_REQUEST:

time = \*(int \*)receivedMessage.body;

pl = std::thread([this, &time, &quit]()

{ pingListener(\*this, time, quit); });

pl.detach();

processPingRequest(time);

break;

case MessageTypes::QUIT:

toSend = exitProcessor(std::move(receivedMessage), quit);

quit = true;

toSend.sendMessage(toOutput);

return;

default:

break;

}

}

}

static void pingListener(SpringBootApplication &that, int &time, bool &exit)

{

while (!exit)

{

for (auto server : that.outerNodes)

{

zmq::socket\_t sender(that.context, zmq::socket\_type::req);

sender.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

Message pingRequest = MessageBuilder::buildPingRequest(time, server.first);

try

{

pingRequest.sendMessage(sender);

if (!pingRequest.receiveMessage(sender, std::chrono::milliseconds(time \* 4)))

{

std::cout << "node " << server.first << " and all her children dead" << std::endl;

// that.outerNodes.erase(server.first);

}

}

catch (const zmq::error\_t &error)

{

continue;

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(time));

}

}

}

void processPingRequest(int time)

{

for (auto server : outerNodes)

{

Message pingRequest = MessageBuilder::buildPingRequest(time, Id);

zmq::socket\_t socket(context, zmq::socket\_type::req);

socket.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

pingRequest.sendMessage(socket);

pingRequest.receiveMessage(socket, std::chrono::milliseconds(100));

// std::cout << "message sent to node " << server.first << std::endl;

socket.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

}

}

void messageOutput()

{

zmq::socket\_t fromMessageProcessor(context, zmq::socket\_type::pull);

fromMessageProcessor.bind(ZmqUtils::MESSAGE\_SENDER\_URL);

Message receivedMessage;

bool quit{false};

while (!quit)

{

receivedMessage.receiveMessage(fromMessageProcessor);

switch (receivedMessage.messageType)

{

case MessageTypes::CREATE\_RESULT:

createOutput(receivedMessage);

break;

case MessageTypes::EXEC\_RESULT:

execOutput(receivedMessage);

break;

case MessageTypes::QUIT:

std::cout << "EXIT" << std::endl;

quit = true;

return;

default:

std::cout << "another message\n";

break;

}

}

}

void ready()

{

std::unique\_lock<std::mutex> lock(creationLock);

created = true;

creationalCondition.notify\_one();

}

void registrator()

{

Message messageIn;

Message messageOut;

while (true)

{

messageIn.receiveMessage(Receiver);

if (messageIn.messageType == MessageTypes::QUIT)

{

messageIn.sendMessage(Receiver);

break;

}

messageOut = processRegisterMessage(messageIn);

ready();

messageOut.sendMessage(Receiver);

}

}

Message processRegisterMessage(Message &message)

{

auto body = (char \*)message.body;

pid\_t pid = \*(pid\_t \*)(body);

int recPort = \*(int \*)(body + sizeof(pid\_t));

int regPort = \*(int \*)(body + sizeof(pid\_t) + sizeof(int));

outerNodes.emplace(message.senderId, ChildNodeInfo(pid, recPort, regPort));

return {MessageTypes::RELATE\_RESULT, Id, message.senderId};

}

Message processCreateMessage(Message &&message)

{

int id = ((int \*)message.body)[0];

int parentId = ((int \*)message.body)[1];

return create(id, parentId);

}

Message create(int id, int pId)

{

if (serversId.find(id) != serversId.end())

{

std::cout << "already exist" << std::endl;

return {MessageTypes::CREATE\_FAIL, Id, Id, "Already exist"};

}

if (serversId.find(pId) == serversId.end())

{

std::cout << "parent not found" << std::endl;

return {MessageTypes::CREATE\_FAIL, Id, Id, "Parent not found"};

}

if (pId == Id)

{

pid\_t pid = addChild(id, Port);

std::cout << "received Pid: " << pid << std::endl;

serversId.insert(id);

std::unique\_lock<std::mutex> lock(creationLock);

created = false;

while (!created)

{

creationalCondition.wait(lock);

}

std::cout << "awaited" << std::endl;

return {MessageTypes::CREATE\_RESULT, Id, Id, sizeof(pid), (void \*)&pid};

}

for (auto server : outerNodes)

{

zmq::socket\_t requestSocket(context, zmq::socket\_type::req);

requestSocket.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

int data[] = {id, pId};

Message request(MessageTypes::CREATE\_REQUEST, Id, server.first, sizeof(data), data);

Message result;

request.sendMessage(requestSocket);

// result.receiveMessage(requestSocket);

if (!result.receiveMessage(requestSocket, std::chrono::milliseconds(2000)))

{

requestSocket.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

continue;

}

if (result.messageType == MessageTypes::CREATE\_RESULT)

{

requestSocket.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

serversId.insert(id);

return {MessageTypes::CREATE\_RESULT, Id, Id, result.sizeOfBody, result.body};

}

requestSocket.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

}

return {MessageTypes::CREATE\_FAIL, Id, Id, "Parent is uavailable"};

}

Message processExecMessage(Message &&message)

{

MessageData data = MessageBuilder::deserialize(message.body);

if (serversId.find(data.id) == serversId.end())

{

std::cout << "id " << data.id << " not found" << std::endl;

return {MessageTypes::EXEC\_FAIL, Id, Id, "Node not found"};

}

for (auto server : outerNodes)

{

zmq::socket\_t request(context, zmq::socket\_type::req);

request.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

message.senderId = Id;

message.recieverId = server.first;

message.sendMessage(request);

Message result;

// result.receiveMessage(request);

if (!result.receiveMessage(request, std::chrono::milliseconds(2000)))

{

request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

continue;

}

if (result.messageType == MessageTypes::EXEC\_RESULT)

{

result.update(Id, Id);

return result;

}

request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

}

std::cout << "node unaviable" << std::endl;

return Message(MessageTypes::EXEC\_FAIL, Id, Id, "node unaviable");

}

Message exitProcessor(Message &&message, bool &exit)

{

zmq::socket\_t exitRegister(context, zmq::socket\_type::req);

exitRegister.connect(ZmqUtils::getOutputAddress(Port));

Message messageExit(MessageTypes::QUIT, Id, Id);

Message temporary;

messageExit.sendMessage(exitRegister);

temporary.receiveMessage(exitRegister);

exitRegister.disconnect(ZmqUtils::getOutputAddress(Port));

std::cout << "closing begins" << std::endl;

exit = true;

for (auto server : outerNodes)

{

zmq::socket\_t request(context, zmq::socket\_type::req);

request.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

messageExit.sendMessage(request);

messageExit.receiveMessage(request, std::chrono::milliseconds(2000));

// messageExit.sendMessage(Receiver);

Message exitResult;

// messageExit.receiveMessage(Receiver, std::chrono::milliseconds(1000));

// exitResult.receiveMessage(request);

request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));

}

std::cout << "servers dead" << std::endl;

return messageExit;

}

void createOutput(Message &message)

{

std::cout << "Ok:" << \*(int \*)message.body << std::endl;

}

void execOutput(Message &message)

{

MessageData data = MessageBuilder::deserialize(message.body);

std::cout << "Exec Ok: " << data.id << " " << data.data[0] << " " << data.data[1] << std::endl;

}

};

**ZmqUtils.h**

#pragma once

#include <string>

#include "zmq.hpp"

class ZmqUtils

{

public:

static const int PORT\_TO\_BIND\_FROM = 30000;

inline static const char \*INPUT\_URL\_TEMPLATE = "tcp://\*:";

inline static const char \*OUTPUT\_URL\_TEMPLATE = "tcp://localhost:";

// in-process communication between threads in running manage node

inline static const char \*MESSAGE\_PROCESSOR\_URL = "inproc://processor";

inline static const char \*MESSAGE\_SENDER\_URL = "inproc://sender";

static std::string getInputAddress(int port)

{

return INPUT\_URL\_TEMPLATE + std::to\_string(port);

}

static std::string getOutputAddress(int port)

{

return OUTPUT\_URL\_TEMPLATE + std::to\_string(port);

}

static int occupyPort(zmq::socket\_t &socket)

{

int currentPort = PORT\_TO\_BIND\_FROM;

while (true)

{

try

{

socket.bind(getInputAddress(currentPort));

break;

}

catch (const zmq::error\_t &error)

{

currentPort++;

}

}

return currentPort;

}

};

**Пример работы**

anatolii@MacBook-Pro-Anatolii ~/D/O/h/l/b/src [SIGINT]> /Users/anatolii/Desktop/Oper\_Syst/hometask\_5\_7/laba/build/src/lab6

create 1 -1

add new node 1 to parent -1

received Pid: 58257

awaited

server created

SERVER CREATED

from registrator 58257

Ok:58257

exec 1 labalaba

lab

1 labalaba lab

0 4

RECEVIEd!! labalaba lab

Exec Ok: 1 0 4

create 2 1

add new node 2 to parent 1

register message got

server created

SERVER CREATED

from registrator 58268

Ok:58268

create 3 1

add new node 3 to parent 1

register message got

server created

SERVER CREATED

from registrator 58273

Ok:58273

create 4 1

add new node 4 to parent 1

register message got

server created

SERVER CREATED

from registrator 58274

Ok:58274

exec 4 popopoababab

poab

4 popopoababab poab

4

RECEVIEd!! popopoababab poab

Exec Ok: 4 4

exec 3 poapoapoapoa

poa

3 poapoapoapoa poa

0 3 6 9

RECEVIEd!! poapoapoapoa poa

Exec Ok: 3 0 3 6 9

exit

closing begins

register message got

servers dead

EXIT

register message got

anatolii@MacBook-Pro-Anatolii ~/D/O/h/l/b/src>

**Вывод**

В ходе работы я познакомился с тем, как можно построить распределенную асинхронную систему с помощью очередей сообщений. Можно добавить, что программа сделана таким образом, что при убийстве какого-нибудь вычислительного узла его потомки само собой станут недоступны, но мы сможем создавать новые узлы и работать уже с созданными и дальше.