МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"
Кафедра 806 "Вычислительная математика и программирование"

Лабораторная работа №5-7

По курсу «Операционные системы»

Студент: Кириллова Е.К.
Группа: М8О-203Б-23
Вариант: 41
Преподаватель: Миронов Е. С.

Дата: _____

Оценка: _____

Подпись: _____

Москва, 2024

Содержание

Репозиторий

https://github.com/ElenaKirillova05/osLabs/tree/main

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.

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

Все вычислительные узлы находятся в списке. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

Набор команд 1 (подсчет суммы n чисел)

Формат команды: exec id n k1 … kn

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

n – количество складываемых чисел (от 1 до 108)

k1 … kn – складываемые числа

Пример:

> exec 10 3 1 2 3

Ok:10: 6

Команда проверки 3

Тип проверки доступности узлов

Команда проверки 1

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

Общие сведения о программе

Общие сведения о программе программа реализует распределенную систему обмена сообщениями между узлами с использованием библиотеки ZeroMQ узлы связаны в бинарное дерево и могут создавать новые узлы выполнять вычисления пересылать сообщения и удалять узлы система состоит из серверной и клиентской частей сервер управляет узлами обрабатывает команды от пользователя и контролирует состояние узлов клиенты представляют собой узлы дерева каждый узел обменивается сообщениями с родительским узлом а также с левым и правым дочерними узлами если они существуют обмен сообщениями реализован асинхронно с использованием ZeroMQ сокетов типа pair для связи между узлами

Общий метод и алгоритм решения

Общий метод и алгоритм решения программа использует архитектуру клиент-сервер для управления узлами в бинарном дереве сервер инициализируется с набором доступных портов и ожидает команды от пользователя команды могут быть следующими создание узла выполнение вычислений в узле проверка доступности узлов ping all и удаление узлов сервер анализирует команду если это команда на создание нового узла он выделяет порт для связи с новым узлом порождает дочерний процесс и запускает клиентскую программу с передачей ей идентификатора узла и выделенного порта если команда касается уже существующего узла сервер пересылает команду соответствующему узлу через сокет клиенты обрабатывают команды в зависимости от их типа если это команда на создание

узла клиент проверяет идентификатор нового узла и определяет будет ли новый узел левым или правым потомком если сокет связи с потомком уже существует клиент пересылает команду по этому сокету если нет то клиент выделяет порт для связи порождает дочерний процесс и запускает новый клиентский узел если команда на выполнение вычислений клиент проверяет соответствует ли идентификатор узла его собственному если да то производит вычисления и отправляет результат родительскому узлу если нет то пересылает команду дочерним узлам если команда на удаление узла клиент закрывает соответствующий сокет и удаляет информацию о дочернем узле сервер поддерживает состояние сети узлов и выполняет проверку их доступности с помощью команды ping all при получении этой команды каждый узел отправляет ответ своему родительскому узлу сервер собирает информацию о доступных узлах определяет недоступные узлы и инициирует их удаление

Исходный код

client.cpp:

```

#include "send_recv.hpp"



void Init(std::uint32_t node_id, std::uint32_t port_id, ClientNode& client_node) {

  client_node.id = node_id;

  client_node.context_client_.emplace();

  client_node.socket_parent_.emplace(client_node.context_client_.value(),
zmq::socket_type::pair);

  client_node.socket_parent_.value().connect(std::string("tcp://localhost:") +
std::to_string(port_id));

}



void Append(std::optional<zmq::socket_t>& node, zmq::context_t& context,
std::optional<std::uint32_t>& setid, std::string message) {

  std::stringstream ss(message);

  std::string create;

  ss >> create;

  std::uint32_t node_id;
```

```
  ss >> node_id;
  std::uint32_t port_id;
  ss >> port_id;
  if (node.has_value()) {
    if (!Send(node.value(), message)) {
      node.reset();
      Append(node, context, setid, message);
    }
  } else {
    node.emplace(context, zmq::socket_type::pair);
    node.value().bind("tcp://*:" + std::to_string(port_id));
    setid.emplace(node_id);
    pid_t pid = fork();
    if (pid == 0) {
      std::string child_id = std::to_string(node_id);
      std::string port = std::to_string(port_id);
      execl("./client", child_id.c_str(), port.c_str(), nullptr);


    }
  }
}

void Create(ClientNode& client_node, std::string message) {
  std::stringstream ss(message);
  std::string create;
  ss >> create;
  std::uint32_t node_id;
  ss >> node_id;
  if (node_id < client_node.id) {
```

```
    Append(client_node.socket_left_, client_node.context_client_.value(), client_node.left_id_, std::move(message));
  } else {
    Append(client_node.socket_right_, client_node.context_client_.value(), client_node.right_id_, std::move(message));
  }
}

void Calculate(ClientNode& client_node, std::string message) {
  std::stringstream ss(message);
  std::string exec;
  ss >> exec;
  std::uint32_t node_id;
  ss >> node_id;
  if (node_id < client_node.id) {
    Send(client_node.socket_left_.value(), std::move(message));
  } else if (node_id > client_node.id) {
    Send(client_node.socket_right_.value(), std::move(message));
  } else {
    std::int32_t ans = 0;
    std::int32_t number = 0;
    while (ss >> number) {
      ans += number;
    }
    Send(client_node.socket_parent_.value(), std::string("OK:") + std::to_string(node_id) + std::string(": ") + std::to_string(ans));
  }
}

void DestroyNode(ClientNode& client_node, std::string message) {
```

```
std::stringstream ss(message);
std::string destroy;
ss >> destroy;
std::uint32_t node_id;
ss >> node_id;
if (node_id < client_node.id) {
  if (client_node.left_id_.has_value()) {
    if (client_node.left_id_.value() == node_id) {
      client_node.left_id_.reset();
      client_node.socket_left_.reset();
    } else {
      if (!Send(client_node.socket_left_.value(), std::move(message))) {
        client_node.left_id_.reset();
        client_node.socket_left_.reset();
      }
    }
  } else {}
} else if (node_id > client_node.id) {
  if (client_node.right_id_.has_value()) {
    if (client_node.right_id_.value() == node_id) {
      client_node.right_id_.reset();
      client_node.socket_right_.reset();
    } else {
      if (!Send(client_node.socket_right_.value(), std::move(message))) {
        client_node.right_id_.reset();
        client_node.socket_right_.reset();
      }
    }
  } else {}
```

```
  }
}
```

```
void ParseAndMakeAction(ClientNode& client_node, std::string message) {
  std::stringstream ss(message);
  std::string command;
  ss >> command;
  if (command == "create") {
    Create(client_node, std::move(message));
  } else if (command == "exec") {
    Calculate(client_node, std::move(message));
  } else if (message == "ping all") {
    Send(client_node.socket_parent_.value(), std::string("node_ping: ") +
std::to_string(client_node.id));
    if (client_node.socket_left_.has_value()) {
      if (!Send(client_node.socket_left_.value(), message)) {
        client_node.socket_left_.reset();
        client_node.socket_right_.reset();
      }
    }
    if (client_node.socket_right_.has_value()) {
      if (!Send(client_node.socket_right_.value(), message)) {
        client_node.socket_right_.reset();
        client_node.right_id_.reset();
      }
    }
  } else if (command == "destroy") {
```

```
    DestroyNode(client_node, std::move(message));
  } else {
    Send(client_node.socket_parent_.value(), std::move(message));
  }
}

#ifndef TEST_BUILD
int main(int argc, char* argv[]) {
  ClientNode client_node;
  pid_t pid = getpid();
  std::uint32_t node_id = std::stoul(argv[0]);
  std::uint32_t port_id = std::stoul(argv[1]);
  Init(node_id, port_id, client_node);
  Send(client_node.socket_parent_.value(), std::string("make node with id: ") +
std::to_string(node_id) + std::string(", pid: ") + std::to_string(pid));
  while (true) {
    while (true) {
      std::optional<std::string> message = Recv(client_node.socket_parent_);
      if (message.has_value()) {
        ParseAndMakeAction(client_node, std::move(message.value()));
      } else {
        break;
      }
    }
    while (true) {
      std::optional<std::string> message = Recv(client_node.socket_left_);
      if (message.has_value()) {
        ParseAndMakeAction(client_node, std::move(message.value()));
      } else {
```

```
      break;
    }
  }
  while (true) {
    std::optional<std::string> message = Recv(client_node.socket_right_);
    if (message.has_value()) {
      ParseAndMakeAction(client_node, std::move(message.value()));
    } else {
      break;
    }
  }
  std::this_thread::sleep_for(0.1s);
 }
}
#endif
```

send_recv.hpp:

```
#pragma once

#include <zmq.hpp>
#include <iostream>
#include <vector>
#include <cstdint>
#include <optional>
#include <sstream>
#include <set>
#include <unistd.h>
```

```
#include <chrono>
#include <thread>
#include <mutex>
#include <algorithm>

using namespace std::chrono_literals;


struct ClientNode {
  std::uint32_t id = 0;
  std::optional<std::uint32_t> left_id_;
  std::optional<std::uint32_t> right_id_;
  std::optional<zmq::context_t> context_client_;
  std::optional<zmq::socket_t> socket_parent_;
  std::optional<zmq::socket_t> socket_left_;
  std::optional<zmq::socket_t> socket_right_;
};


struct ServerNode {
  bool check_ping_ = false;
  std::uint32_t id = 0;
  std::uint32_t child_id = 0;
  std::optional<zmq::socket_t> socket_root_;
  std::optional<zmq::context_t> context_root_;
  std::vector<std::pair<std::uint32_t, bool>> ports_;
  std::set<std::uint32_t> node_id_;
};
```

```
bool Send(zmq::socket_t& receiver, std::string message) {
  zmq::message_t request(message.size());
  std::memcpy(request.data(), message.c_str(), message.size());
  if (receiver.send(request, zmq::send_flags::dontwait).has_value()) {
    return true;
  }
  return false;
}


std::optional<std::string> Recv(std::optional<zmq::socket_t>& sender) {
  if (sender.has_value()) {
    zmq::message_t message;
    std::optional<size_t> count_bytes = sender.value().recv(message, zmq::recv_flags::dontwait);
    if (count_bytes.has_value()) {
      return {std::string(static_cast<char*>(message.data()), message.size())};
    }
  }
  return std::nullopt;
}
```

server.cpp:

```
#include "send_recv.hpp"

std::optional<std::uint32_t> GetPort(std::vector<std::pair<std::uint32_t, bool>>& ports) {
```

```
  std::uint32_t ans = 0;
  for (auto& item : ports) {
    if (item.second) {
      item.second = false;
      ans = item.first;
      break;
    }
  }
  if (ans == 0) {
    return std::nullopt;
  }
  return {ans};
}

bool InServer(ServerNode& node, std::uint32_t id) {
  return node.node_id_.find(id) != node.node_id_.end();
}

void InitPorts(std::vector<std::pair<std::uint32_t, bool>>& ports, std::uint32_t start_port) {
  for (auto& item : ports) {
    item.first = start_port++;
    item.second = true;
  }
}

bool ExecParse(ServerNode& server_node, std::string message) {
  std::stringstream ss(message);
  std::string exec;
  ss >> exec;
```

```
  std::uint32_t node_id = 0;
  ss >> node_id;
  if (InServer(server_node, node_id)) {
    Send(server_node.socket_root_.value(), std::move(message));
    return true;
  }
  return false;
}

void Init(ServerNode& server_node) {
  server_node.context_root_.emplace();
  server_node.ports_.resize(100, {0, true});
  InitPorts(server_node.ports_, 1070);
  server_node.node_id_.emplace(0);
}

bool Create(ServerNode& server_node, std::string message) {
  std::stringstream ss(message);
  std::string create;
  ss >> create;
  std::uint32_t node_id = 0;
  ss >> node_id;
  if (InServer(server_node, node_id)) {
    return false;
  }
  std::optional<std::uint32_t> port_id = GetPort(server_node.ports_);
  if (!port_id.has_value()) {
    return false;
  }
```

```
  server_node.node_id_.emplace(node_id);
  if (server_node.socket_root_.has_value()) {
    if (!Send(server_node.socket_root_.value(), message + ' ' + std::to_string(port_id.value()))) {
      server_node.socket_root_.reset();
      return Create(server_node, std::move(message));
    }
  } else {
    server_node.socket_root_.emplace(server_node.context_root_.value(),
zmq::socket_type::pair);
    server_node.socket_root_.value().bind("tcp://*:" + std::to_string(port_id.value()));
    server_node.child_id = node_id;
    pid_t pid = fork();
    if (pid == 0) {
      std::string child_id = std::to_string(node_id);
      std::string port = std::to_string(port_id.value());
      execl("./client", child_id.c_str(), port.c_str(), nullptr);
    }
  }
  return true;
}

void AnalizeSenders(ServerNode& server_node, std::string message);

void ReceivQueue(ServerNode& server_node, std::vector<std::uint32_t>& availible_nodes) {
  while (true) {
    std::optional<std::string> message = Recv(server_node.socket_root_);
    if (message.has_value()) {
      std::stringstream ss(message.value());
      std::string command;
```

```
      ss >> command;
      if (command != "node_ping:") {
        AnalizeSenders(server_node, std::move(message.value()));
      } else {
        std::uint32_t node_id;
        ss >> node_id;
        availible_nodes.push_back(node_id);
      }
    } else {
      break;
    }
  }
}

std::vector<std::uint32_t> GetUnavailibleNodes(ServerNode& server_node, const
std::vector<std::uint32_t>& avail_nodes) {
  std::vector<std::uint32_t> ans_;
  for (auto item : server_node.node_id_) {
    if (item == 0) continue;
    if (std::find(avail_nodes.begin(), avail_nodes.end(), item) == avail_nodes.end()) {
      ans_.push_back(item);
    }
  }
  return ans_;
}

void Destroy(ServerNode& server_node, const std::vector<std::uint32_t>& destroy_nodes) {
  for (auto item : destroy_nodes) {
    server_node.node_id_.erase(item);
```

```
  }
}

void CallDestroyResponse(ServerNode& server_node, const std::vector<std::uint32_t>& destroy_nodes) {
  for (auto item : destroy_nodes) {
    std::cout << item << ' ';
    if (!Send(server_node.socket_root_.value(), std::string("destroy ") + std::to_string(item))) {
      server_node.socket_root_.reset();
      server_node.child_id = 0;
    }
  }
}

void AnalizeSenders(ServerNode& server_node, std::string message) {
  std::stringstream ss(message);
  std::string command;
  ss >> command;
  if (command == std::string("node_ping:")) {
    if (server_node.check_ping_ == true) {
      std::uint32_t node_id;
      ss >> node_id;
      std::vector<std::uint32_t> availible_nodes;
      availible_nodes.push_back(node_id);
      ReceivQueue(server_node, availible_nodes);
      std::vector<std::uint32_t> unavailible_nodes_ = GetUnavailibleNodes(server_node, availible_nodes);
      Destroy(server_node, unavailible_nodes_);
      CallDestroyResponse(server_node, unavailible_nodes_);
```

```
      server_node.check_ping_ = false;
      std::cout << "Availible nodes: ";
      for (auto item : availible_nodes) {
        std::cout << item << ' ';
      }
      std::cout << std::endl;
    } else {}
  } else {
    std::cout << message << std::endl;
  }
}

void AnalizeUser(ServerNode& server_node, std::string message) {
  std::stringstream ss(message);
  std::string command;
  ss >> command;
  if (command == std::string("create")) {
    if (!Create(server_node, std::move(message))) {
      std::cout << "Max limit or node already in system" << std::endl;
    }
  } else if (command == "exec") {
    if (!ExecParse(server_node, std::move(message))) {
      std::cout << "Node not in system!" << std::endl;
    }
  } else if (message == "ping all") {
    server_node.check_ping_ = true;
    if (!server_node.socket_root_.has_value() || !Send(server_node.socket_root_.value(),
message)) {
      server_node.socket_root_.reset();
```

```
      server_node.child_id = 0;
      server_node.check_ping_ = false;
      server_node.node_id_.clear();
      server_node.node_id_.emplace(0);
      return;
    }
    std::cout << "Please wait. Nodes checking..." << std::endl;
    std::this_thread::sleep_for(1s);
  } else {
    std::cout << "Incorrect command" << std::endl;
  }
}

#ifndef TEST_BUILD
int main() {
  ServerNode server_node;
  Init(server_node);
  std::string user_command;
  while (true) {
    while (true) {
      std::optional<std::string> message = Recv(server_node.socket_root_);
      if (message.has_value()) {
        AnalizeSenders(server_node, std::move(message.value()));
      } else {
        break;
      }
    }
    std::getline(std::cin, user_command);
    AnalizeUser(server_node, std::move(user_command));
```

```
  }
}
#endif
```

ТЕСТЫ

tests.cpp:
```
#define TEST_BUILD

#include <gtest/gtest.h>
#include "send_recv.hpp"
#include "server.cpp"
#include "client.cpp"

TEST(ClientNodeTest, InitTest) {
    ClientNode client_node;
    Init(1, 1070, client_node);

    EXPECT_EQ(client_node.id, 1);

    EXPECT_TRUE(client_node.context_client_.has_value());
    EXPECT_TRUE(client_node.socket_parent_.has_value());
}

TEST(ClientNodeTest, AppendTest) {
    zmq::context_t context;
```

```
    std::optional<zmq::socket_t> socket;
    std::optional<std::uint32_t> node_id;
    std::string message = "create 2";

    Append(socket, context, node_id, message);

    EXPECT_TRUE(socket.has_value());
    EXPECT_TRUE(node_id.has_value());
    EXPECT_EQ(node_id.value(), 2);
}

TEST(ServerNodeTest, InitTest) {
    ServerNode server_node;
    Init(server_node);

    EXPECT_EQ(server_node.node_id_.size(), 1);

    EXPECT_EQ(server_node.ports_.size(), 100);
}

TEST(ServerNodeTest, PingAllTest) {
    ServerNode server_node;
    Init(server_node);
    std::string message = "ping all";

    try {
        EXPECT_NO_THROW({
            if (server_node.node_id_.empty()) {
```

```
            FAIL() << "Node ID set is empty!";
        }
        AnalizeUser(server_node, message);
    });
  } catch (const std::bad_optional_access& e) {
    FAIL() << "Caught bad_optional_access exception: " << e.what();
  } catch (const std::exception& e) {
    FAIL() << "Caught unexpected exception: " << e.what();
  }
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Вывод

Разработанная программа реализует распределенную систему обмена сообщениями между узлами организованными в виде бинарного дерева с использованием библиотеки zeromq асинхронный обмен сообщениями и архитектура клиент-сервер обеспечивают гибкость и масштабируемость системы программа позволяет динамически создавать удалять узлы выполнять вычисления и проверять доступность узлов подход с использованием сокетов типа pair и механизмов zeromq обеспечивает надежную передачу сообщений и устойчивость системы к сбоям архитектура позволяет легко расширять функциональность и адаптировать систему под различные задачи распределенных вычислений