Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторные работы №5-7 по курсу «Операционные системы»

Студент: Велиев Рауф Рамиз оглы
Группа: М8О-209Б-23
Вариант: 4
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

Содержание

Репозиторий

https://github.com/velievrauf/OS/tree/main/lab5-7

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, исполнение команды на вычислительном узле.

Тип топологии: Топология 2 (аналогично Топологии 1, но узлы находятся в дереве общего вида);

Тип команд для вычислительных узлов: Набор команд 3 (локальный таймер);

Тип проверки доступности узлов: Команда проверки 3 (heartbit time).

Общий метод и алгоритм решения

1. Инициализация системы:
    - Создаем управляющий узел, который будет принимать команды от пользователя и взаимодействовать с вычислительными узлами через очередь сообщений.
2. Создание и управление узлами:
    - Управляющий узел создает вычислительные узлы в соответствии с заданной топологией (дерево общего вида). Узлы создаются через команды create, передаваемые по сети.
3. Обмен сообщениями:
    - Для связи между узлами используем технологии очередей сообщений (ZeroMQ).
    - Команды, отправленные с управляющего узла, передаются дочерним узлам с помощью сообщений.
4. Обработка команд:
    - **Команды выполнения**: Каждый вычислительный узел реализует локальный таймер с командами start, stop, time.
    - **Проверка доступности**: Реализуем механизм heartbeat с определенным интервалом (heartbit time), чтобы контролировать состояние узлов.
5. Устойчивость системы:
    - При завершении работы любого вычислительного узла его дочерние узлы становятся недоступны, но остальные узлы продолжают функционировать.
6. Механизм удаления узлов:

- Команда remove удаляет указанный узел и все его дочерние узлы, с корректным закрытием соединений.

Исходный код

CMakeLists.txt

```
# Указываем имя проекта
project(ZMQProject)

# Указываем стандарт C++
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Параметры компиляции
include_directories(/opt/homebrew/include)
include_directories(${CMAKE_SOURCE_DIR})  # Добавляем текущую директорию для заголовков
link_directories(/opt/homebrew/lib)

# Основные цели: клиент и воркер
add_executable(client client.cpp)
add_executable(worker worker.cpp)

# Линковка ZeroMQ
target_link_libraries(client zmq)
target_link_libraries(worker zmq)

# Подключаем Google Test
find_package(GTest REQUIRED)    # Найти установленный Google Test
find_package(Threads REQUIRED) # Подключаем поддержку потоков
include_directories(${GTEST_INCLUDE_DIRS})

# Создаём цель для тестов
add_executable(tests
    tests/test_main.cpp        # Файл с тестами
    node.h                  # Заголовочные файлы
    net_func.h
)

# Линкуем тесты с Google Test, ZeroMQ и потоками
target_link_libraries(tests PRIVATE ${GTEST_LIBRARIES} zmq Threads::Threads)

# Включаем тестирование
enable_testing()
add_test(NAME ZMQTests COMMAND tests)

# Команда для очистки
add_custom_target(clean COMMAND ${CMAKE_COMMAND} -E remove -f client worker tests)
```

client.cpp

```
#include "node.h"
#include "net_func.h"
#include "set"
#include <signal.h>
#include <chrono>
#include <thread>
```

```
#include <mutex>
#include <atomic>
#include <map>

static std::atomic<bool> heartbeat_active(false);
static std::atomic<int> heartbeat_interval_ms(0);
static std::thread heartbeat_thread;
static bool heartbeat_thread_started = false;
static std::mutex nodes_mutex;
static std::map<int, std::chrono::time_point<std::chrono::steady_clock>> last_success_ping;

void heartbeat_thread_func(Node* me, std::set<int>* all_nodes) {
   // Поток отправляет пинги всем известным узлам каждые heartbeat_interval_ms миллисекунд
   // и проверяет, есть ли узлы, не ответившие уже > 4 раза дольше интервала.
   while (heartbeat_active.load()) {
      int interval = heartbeat_interval_ms.load();
      if (interval <= 0) {
         std::this_thread::sleep_for(std::chrono::milliseconds(100));
         continue;
      }

      {
         std::lock_guard<std::mutex> lock(nodes_mutex);
         for (auto node_id : *all_nodes) {
            if (node_id == -1) {
               // Корневой узел - скип
               continue;
            }
            // Пингуем узел
            std::string ans;
            if (me->children.find(node_id) != me->children.end()) {
               ans = me->Ping_child(node_id);
            } else {
               std::string str = "ping " + std::to_string(node_id);
               ans = me->Send(str, node_id);
               if (ans == "Error: not find") {
                  ans = "Ok: 0";
               }
            }

            if (ans == "Ok: 1") {
               // Узел доступен, обновляем время последнего успешного ответа
               last_success_ping[node_id] = std::chrono::steady_clock::now();
            } else {
               // Узел недоступен или не найден. Если его нет в last_success_ping, занесем текущее время,
               // чтобы отсчитать таймер недоступности.
               if (last_success_ping.find(node_id) == last_success_ping.end()) {
                  last_success_ping[node_id] = std::chrono::steady_clock::now();
               } else {
                  // Проверим, сколько времени прошло с последнего успешного пинга
                  auto now = std::chrono::steady_clock::now();
                  auto diff = std::chrono::duration_cast<std::chrono::milliseconds>(now -
last_success_ping[node_id]).count();
                  // Если прошло более чем 4 * interval, сообщаем о недоступности
                  if (diff > 4 * interval) {
                     std::cout << "Heartbit: node " << node_id << " is unavailable now" << std::endl;
                     // Чтобы сообщение не повторялось бесконечно, обновим время так, чтобы снова ждать
                     last_success_ping[node_id] = std::chrono::steady_clock::now();
                  }
               }
```

```
            }
        }
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(heartbeat_interval_ms.load()));
  }
}

int main() {
  std::set<int> all_nodes;
  std::string prog_path = "./worker";
  Node me(-1);
  all_nodes.insert(-1);
  last_success_ping[-1] = std::chrono::steady_clock::now();
  std::string command;
  while (std::cin >> command) {
    if (command == "create") {
      int id_child, id_parent;
      std::cin >> id_child >> id_parent;
      if (all_nodes.find(id_child) != all_nodes.end()) {
        std::cout << "Error: Already exists" << std::endl;
      } else if (all_nodes.find(id_parent) == all_nodes.end()) {
        std::cout << "Error: Parent not found" << std::endl;
      } else if (id_parent == me.id) {
        std::string ans = me.Create_child(id_child, prog_path);
        std::cout << ans << std::endl;
        all_nodes.insert(id_child);
      } else {
        std::string str = "create " + std::to_string(id_child);
        std::string ans = me.Send(str, id_parent);
        std::cout << ans << std::endl;
        all_nodes.insert(id_child);
      }
    } else if (command == "ping") {
      int id_child;
      std::cin >> id_child;
      if (all_nodes.find(id_child) == all_nodes.end()) {
        std::cout << "Error: Not found" << std::endl;
      } else if (me.children.find(id_child) != me.children.end()) {
        std::string ans = me.Ping_child(id_child);
        std::cout << ans << std::endl;
      } else {
        std::string str = "ping " + std::to_string(id_child);
        std::string ans = me.Send(str, id_child);
        if (ans == "Error: not find") {
          ans = "Ok: 0";
        }
        std::cout << ans << std::endl;
      }
    } else if (command == "exec") {
      int id;
      std::string cmd;
      std::cin >> id >> cmd;
      std::string msg = "exec " + cmd;
      if (all_nodes.find(id) == all_nodes.end()) {
        std::cout << "Error: Not found" << std::endl;
      } else {
        std::string ans = me.Send(msg, id);
        std::cout << ans << std::endl;
      }
```

```
    } else if (command == "remove") {
        int id;
        std::cin >> id;
        std::string msg = "remove";
        if (all_nodes.find(id) == all_nodes.end()) {
            std::cout << "Error: Not found" << std::endl;
        } else {
            std::string ans = me.Send(msg, id);
            if (ans != "Error: not find") {
                std::istringstream ids(ans);
                int tmp;
                while (ids >> tmp) {
                    all_nodes.erase(tmp);
                }
                ans = "Ok";
                if (me.children.find(id) != me.children.end()) {
                    my_net::unbind(me.children[id], me.children_port[id]);
                    me.children[id]->close();
                    me.children.erase(id);
                    me.children_port.erase(id);
                }
            }
            std::cout << ans << std::endl;
        }
        }else if (command == "heartbit") {
            int time_ms;
            std::cin >> time_ms;
            if (time_ms <= 0) {
                std::cout << "Error: invalid time" << std::endl;
                continue;
            }
            heartbeat_interval_ms.store(time_ms);
            if (!heartbeat_thread_started) {
                heartbeat_active.store(true);
                heartbeat_thread = std::thread(heartbeat_thread_func, &me, &all_nodes);
                heartbeat_thread.detach();
                heartbeat_thread_started = true;
            }
            std::cout << "Ok" << std::endl;
        }
    }

    heartbeat_active.store(false);
    me.Remove();
    return 0;
    }
```

net_func.h

```
#pragma once

#include <iostream>
#include <zmq.hpp>
#include <sstream>
#include <string>

namespace my_net {

#define MY_PORT 4040
#define MY_IP "tcp://127.0.0.1:"
```

```
    int bind(zmq::socket_t *socket, int id) {
        int port = MY_PORT + id;
        while (true) {
            std::string adress = MY_IP + std::to_string(port);
            try {
                socket->bind(adress);
                break;
            } catch (...) {
                port++;
            }
        }
        return port;
    }

    void connect(zmq::socket_t *socket, int port) {
        std::string adress = MY_IP + std::to_string(port);
        socket->connect(adress);
    }

    void unbind(zmq::socket_t *socket, int port) {
        std::string adress = MY_IP + std::to_string(port);
        socket->unbind(adress);
    }

    void disconnect(zmq::socket_t *socket, int port) {
        std::string adress = MY_IP + std::to_string(port);
        socket->disconnect(adress);
    }

    void send_message(zmq::socket_t *socket, const std::string msg) {
        zmq::message_t message(msg.size());
        memcpy(message.data(), msg.c_str(), msg.size());
        try {
            socket->send(message);
        } catch (...) {}
    }

    std::string reseave(zmq::socket_t *socket) {
        zmq::message_t message;
        bool success = true;
        try {
            socket->recv(&message, 0);
        } catch (...) {
            success = false;
        }
        if (!success || message.size() == 0) {
            throw -1;
        }
        std::string str(static_cast<char *>(message.data()), message.size());
        return str;
    }

}
```

node.h

```
#include <iostream>
#include "net_func.h"
```

```
#include <sstream>
#include <unordered_map>
#include "unistd.h"

class Node {
private:
    zmq::context_t context;
public:
    std::unordered_map<int, zmq::socket_t *> children;
    std::unordered_map<int, int> children_port;
    zmq::socket_t parent;
    int parent_port;
    int id;

    Node(int _id, int _parent_port = -1) : parent(context, ZMQ_REP),
                                           parent_port(_parent_port),
                                           id(_id) {
        if (_id != -1) {
            my_net::connect(&parent, _parent_port);
        }
    }

    std::string Ping_child(int _id) {
        std::string ans = "Ok: 0";
        if (_id == id) {
            ans = "Ok: 1";
            return ans;
        } else if (children.find(_id) != children.end()) {
            std::string msg = "ping " + std::to_string(_id);
            my_net::send_message(children[_id], msg);
            try {
                msg = my_net::reseave(children[_id]);
                if (msg == "Ok: 1")
                    ans = msg;
            } catch (int) {}
            return ans;
        } else {
            return ans;
        }
    }

    std::string Create_child(int child_id, std::string program_path) {
        std::string program_name = program_path.substr(program_path.find_last_of("/") + 1);
        children[child_id] = new zmq::socket_t(context, ZMQ_REQ);

        int new_port = my_net::bind(children[child_id], child_id);
        children_port[child_id] = new_port;
        int pid = fork();

        if (pid == 0) {
            execl(program_path.c_str(), program_name.c_str(), std::to_string(child_id).c_str(),
                  std::to_string(new_port).c_str(), (char *) NULL);
        } else {
            std::string child_pid;
            try {
                children[child_id]->setsockopt(ZMQ_SNDTIMEO, 3000);
                my_net::send_message(children[child_id], "pid");
                child_pid = my_net::reseave(children[child_id]);
            } catch (int) {
                child_pid = "Error: can't connect to child";
```

```
        }
        return "Ok: " + child_pid;
    }
}

std::string Pid() {
    return std::to_string(getpid());
}

std::string Send(std::string str, int _id) {
    if (children.size() == 0) {
        return "Error: now find";
    } else if (children.find(_id) != children.end()) {
        if (Ping_child(_id) == "Ok: 1") {
            my_net::send_message(children[_id], str);
            std::string ans;
            try {
                ans = my_net::reseave(children[_id]);
            } catch (int) {
                ans = "Error: now find";
            }
            return ans;
        }
    } else {
        std::string ans = "Error: not find";
        for (auto &child: children) {
            if (Ping_child(child.first) == "Ok: 1") {
                std::string msg = "send " + std::to_string(_id) + " " + str;
                my_net::send_message(children[child.first], msg);
                try {
                    msg = my_net::reseave(children[child.first]);
                } catch (int) {
                    msg = "Error: not find";
                }
                if (msg != "Error: not find") {
                    ans = msg;
                }
            }
        }
        return ans;
    }
    return "Error: not find";
}

std::string Remove() {
    std::string ans;
    if (children.size() > 0) {
        for (auto &child: children) {
            if (Ping_child(child.first) == "Ok: 1") {
                std::string msg = "remove";
                my_net::send_message(children[child.first], msg);
                try {
                    msg = my_net::reseave(children[child.first]);
                    if (ans.size() > 0)
                        ans = ans + " " + msg;
                    else
                        ans = msg;
                } catch (int) {}
            }
            my_net::unbind(children[child.first], children_port[child.first]);
```

```
            children[child.first]->close();
        }
        children.clear();
        children_port.clear();
    }
    return ans;
  }
};
```

worker.cpp

```
#include "node.h"
#include "net_func.h"
#include <fstream>
#include <vector>
#include <signal.h>
#include <chrono>

int my_id = 0;

static bool timer_running = false;
static std::chrono::time_point<std::chrono::steady_clock> start_time;
static std::chrono::duration<double> elapsed(0.0);

void Log(std::string str) {
  std::string f = std::to_string(my_id) + ".txt";
  std::ofstream fout(f, std::ios_base::app);
  fout << str;
  fout.close();
}

int main(int argc, char **argv) {
  if (argc != 3) {
    return -1;
  }

  Node me(atoi(argv[1]), atoi(argv[2]));
  my_id = me.id;
  std::string prog_path = "./worker";
  while (1) {
    std::string message;
    std::string command = " ";
    message = my_net::reseave(&(me.parent));
    std::istringstream request(message);
    request >> command;

    if (command == "create") {
      int id_child;
      request >> id_child;
      std::string ans = me.Create_child(id_child, prog_path);
      my_net::send_message(&me.parent, ans);
    } else if (command == "pid") {
      std::string ans = me.Pid();
      my_net::send_message(&me.parent, ans);
    } else if (command == "ping") {
      int id_child;
      request >> id_child;
      std::string ans = me.Ping_child(id_child);
      my_net::send_message(&me.parent, ans);
```

```
        } else if (command == "send") {
            int id;
            request >> id;
            std::string str;
            getline(request, str);
            str.erase(0, 1);
            std::string ans;
            ans = me.Send(str, id);
            my_net::send_message(&me.parent, ans);
        } else if (command == "exec") {
            std::string cmd;
            request >> cmd;
            std::string ans;
            if (cmd == "start") {
                if (!timer_running) {
                    timer_running = true;
                    elapsed = std::chrono::duration<double>(0);
                    start_time = std::chrono::steady_clock::now();
                    ans = "Ok:" + std::to_string(me.id) + ":timer started";
                } else {
                    ans = "Ok:" + std::to_string(me.id) + ":timer already running";
                }
            } else if (cmd == "time") {
                if (timer_running) {
                    auto now = std::chrono::steady_clock::now();
                    auto diff = std::chrono::duration_cast<std::chrono::milliseconds>(now - start_time).count();
                    ans = "Ok:" + std::to_string(me.id) + ":elapsed " + std::to_string(diff) + " ms";
                } else {
                    ans = "Ok:" + std::to_string(me.id) + ":timer not running";
                }
            } else if (cmd == "stop") {
                if (timer_running) {
                    auto now = std::chrono::steady_clock::now();
                    auto diff = std::chrono::duration_cast<std::chrono::milliseconds>(now - start_time).count();
                    timer_running = false;
                    ans = "Ok:" + std::to_string(me.id) + ":timer stopped at " + std::to_string(diff) + " ms";
                } else {
                    ans = "Ok:" + std::to_string(me.id) + ":timer not running";
                }
            } else {
                ans = "Error: invalid exec command";
            }
            my_net::send_message(&me.parent, ans);
        } else if (command == "remove") {
    std::cout << "[WORKER] Removing node " << my_id << "..." << std::endl;
    std::string ans = me.Remove();
    ans = std::to_string(me.id) + " " + ans;
    my_net::send_message(&me.parent, ans);
    my_net::disconnect(&me.parent, me.parent_port);
    me.parent.close();
    std::cout << "[WORKER] Node " << my_id << " removed successfully." << std::endl;
    break;
}
    }
    sleep(1);
    return 0;
}
```

Демонстрация работы программы

create 10 -1

Ok: 3181

ping 10

Ok: 1

remove 10

[WORKER] Removing node 10…

[WORKER] Node 10 removed successfully.

Ok

Выводы

В процессе лабораторной работы была разработана распределённая система для асинхронной обработки запросов с применением ZeroMQ. В основе решения лежит деревообразная топология узлов, которая обеспечивает гибкость масштабирования и надёжность системы. Для мониторинга состояния узлов реализован механизм heartbeat, позволяющий оперативно обнаруживать сбои и отключение компонентов.

Система успешно выполняет обработку задач и демонстрирует устойчивость к отказам, продолжая функционировать даже при выходе из строя отдельных узлов. В рамках реализации предусмотрены команды управления, включая создание, удаление, проверку состояния узлов и выполнение задач.

Проведённое тестирование подтвердило эффективность и стабильность разработанного решения, а также его способность адаптироваться к изменениям нагрузки и инфраструктуры.