Московский авиационный институт
(Национальный исследовательский университет)

Институт: №8 «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Операционные системы»

Лабораторные работы № 6-8

Тема: Управлении серверами сообщений. Применение отложенных вычислений. Интеграция программных систем друг с другом

Студент: Бирюков В. В.
Группа: М80-207Б-19
Преподаватель: Миронов Е. С.
Дата:
Оценка:

Москва, 2020

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 вычислительного узла
- Удаление существующего вычислительного узла
- Исполнение команды на вычислительном узле

Вариант 4.

Топология 1 – все вычислительные узлы находятся в списке. Есть только один управляющий узел.

Набора команд 2 – локальный целочисленный словарь.

Команда проверки 1 – вывод всех недоступных узлов.

Алгоритм решения задачи

Для реализации взаимодействия узлов выбрана библиотека ZMQ.

Каждый вычислительный узел связан с предыдущим и следующим. Каждая связь состоит из пары односторонних связей, реализованных при помощи сокетов PUSH/PULL. Управляющий узел хранит карту сети в виде списка списков, а также исходящие связи с узлами первого уровня и одну входящую с этими же узлами.

Проверка доступности узлов не является асинхронным действием, поэтому каждый узел связан с предыдущим и следующим парой сокетов REQ/REP, причем сокетам, связанным со следующим узлом задано время ожидания получения ответа, что позволяет обнаруживать недоступные узлы.

Взаимодействие между узлами организовано при помощи команд, помещенных в перечислимый тип. Все сообщение состоит из идентификатора адресата, команды и дополнительных аргументов. Если узел получает сообщение с другим идентификатором, он отправляет его дальше.

Листинг программы

```
// main.cpp
// Управляющий узел
#include <unistd.h>
#include <pthread.h>
#include <zmqpp/zmqpp.hpp>
#include <iostream>
#include <list>
#include <string>
#include <signal.h>
#include <sstream>

#include "network.hpp"

using line_t = std::pair<std::list<int>, std::pair<std::pair<zmqpp::socket,
std::string>, std::pair<zmqpp::socket, std::string>>>;
std::list<line_t> network;

class node_coord {
private:
	std::pair<std::list<line_t>::iterator, std::list<int>::iterator> data;

public:
	node_coord(std::list<line_t>::iterator& i1, std::list<int>::iterator& i2)
: data(i1, i2) {}
	node_coord(std::list<line_t>::iterator&& i1, std::list<int>::iterator&&
i2) : data(i1, i2) {}
	node_coord(node_coord& other) : data(other.data) {}
	node_coord(node_coord&& other) : data(other.data) {}

	std::list<line_t>::iterator& line() {
		return data.first;
	}

	std::list<int>::iterator& node() {
		return data.second;
	}

	int& id() {
		return *(data.second);
	}

	int& parent() {
		// main parent
		return data.first->first.front();
	}

	zmqpp::socket& out_sock() {
		return data.first->second.first.first;
	}

	zmqpp::socket& ping_sock() {
		return data.first->second.second.first;
	}

	std::string& out_port() {
		return data.first->second.first.second;
	}

	std::string& ping_port() {
		return data.first->second.second.second;
	}
};
```

```
node_coord find_node(int id) {
	for (auto b_it = network.begin(); b_it != network.end(); ++b_it) {
		for (auto i_it = b_it->first.begin(); i_it != b_it->first.end(); +
+i_it) {
			if (*i_it == id) {
				return {b_it, i_it};
			}
		}
	}
	return {network.end(), std::list<int>::iterator()};
}

void* result_waiter(void* arg) {
	zmqpp::socket *in_sock = reinterpret_cast<zmqpp::socket*>(arg);
	zmqpp::message msg;
	int rid, act;
	while (in_sock) {
		in_sock->receive(msg);
		msg >> rid >> act;

		switch (static_cast<action>(act)) {
		case action::fork: {
			int pid;
			msg >> pid;
			std::cout << "OK: " << pid << std::endl;
			break;
		}

		case action::exec_set: {
			std::cout << "OK:" << rid << std::endl;
			break;
		}

		case action::exec_get: {
			std::string name, value;
			msg >> name >> value;
			if (value == "not found") {
				std::cout << "OK:" << rid << ": '" << name << "' not
found" << std::endl;
			} else {
				std::cout << "OK:" << rid << ": " << value << std::endl;
			}
			break;
		}

		case action::rebind_back: {
			node_coord node = find_node(rid);
			node.out_sock().disconnect(host + node.out_port());
			msg >> node.out_port();
			node.out_sock().connect(host + node.out_port());

			node.ping_sock().disconnect(host + node.ping_port());
			msg >> node.ping_port();
			node.ping_sock().connect(host + node.ping_port());
			break;
		}

		case action::exit: {
			node_coord node = find_node(rid);
			node.line()->first.erase(node.node());
			if (node.line()->first.empty()) {
				network.erase(node.line());
			}
```

```
            }

            default: {}

            }
      }
      return NULL;
}

zmqpp::context context;

bool ping(zmqpp::socket& ping_sock, std::string& ping_port, int id) {
      int packet[2] = {id, 1};
      size_t length = 2 * sizeof(int);
      if (!ping_sock.send_raw(reinterpret_cast<char *>(packet), length,
zmqpp::socket::dont_wait)) {
            return false;
      }
      if (!ping_sock.receive_raw(reinterpret_cast<char *>(packet), length)) {
            ping_sock.close();
            ping_sock = zmqpp::socket(context, zmqpp::socket_type::req);
            ping_sock.set(zmqpp::socket_option::receive_timeout, 1000);
            ping_sock.connect(host + ping_port);
            return false;
      }
      return (packet[1] == 1) ? true : false;
}

int main() {
      // std::cout << getpid() << std::endl;


      zmqpp::socket in_sock(context, zmqpp::socket_type::pull);
      std::string in_port = std::to_string(try_bind(in_sock));

      zmqpp::socket bridge(context, zmqpp::socket_type::rep);
      std::string bridge_port = std::to_string(try_bind(bridge));

      pthread_t result_waiter_id;
      check(pthread_create(&result_waiter_id, NULL, result_waiter,
(void*)&in_sock), -1, "pthread_create error");
      if (pthread_detach(result_waiter_id) != 0) {
            perror("detach error");
      }

      std::string command;
      std::cout << "> ";
      while (std::cin >> command && command != "exit") {
            if (command == "create") {
                  int id, parent;
                  std::cin >> id >> parent;

                  if (id == -1 || find_node(id).line() != network.end()) {
                        std::cerr << "Error: Already exists" << std::endl;
                        continue;
                  }

                  if (parent == -1) {
                        network.push_back({std::list<int>(),
std::make_pair(std::make_pair(zmqpp::socket(context, zmqpp::socket_type::push),
""), std::make_pair(zmqpp::socket(context, zmqpp::socket_type::req), ""))});
                        network.back().first.push_back(id);
                        node_coord node(--network.end(), --
network.back().first.end());
```

```
                        int pid = fork();
                        check(id, -1, "fork error");
                        if (pid == 0) {
                              check(execl("node", "node",
std::to_string(id).c_str(), std::to_string(-1).c_str(),
                                                      bridge_port.c_str(), NULL), -1,
"execl error")
                        }

                        zmqpp::message ports;
                        bridge.receive(ports);

                        ports >> node.out_port();
                        node.out_sock().connect(host + node.out_port());

                        ports >> node.ping_port();
                        node.ping_sock().connect(host + node.ping_port());

      node.ping_sock().set(zmqpp::socket_option::receive_timeout, 1000);

                        ports.pop_back();
                        ports.pop_back();
                        ports << in_port;
                        bridge.send(ports);
                  }

                  else {
                        node_coord parent_node = find_node(parent);
                        if (parent_node.line() == network.end()) {
                              std::cerr << "Error: Parent not found" <<
std::endl;
                              continue;
                        }

                        if (!ping(parent_node.ping_sock(),
parent_node.ping_port(), parent)) {
                              std::cerr << "Error: Parent is unavailable" <<
std::endl;
                              continue;
                        }

                        parent_node.line()->first.insert(++parent_node.node(),
id);

                        zmqpp::message msg;
                        msg << parent << static_cast<int>(action::fork) << id;
                        parent_node.out_sock().send(msg);
                  }
            }

            else if (command == "remove") {
                  int id;
                  std::cin >> id;

                  node_coord node = find_node(id);

                  if (node.line() == network.end()) {
                        std::cerr << "Error: Not found" << std::endl;
                        continue;
                  }

                  if (!ping(node.ping_sock(), node.ping_port(), id)) {
                        std::cerr << "Error: Node is unavailable" << std::endl;
```

```
                        continue;
                }

                zmqpp::message msg;
                msg << id << static_cast<int>(action::exit);
                node.out_sock().send(msg);
        }

        else if (command == "exec") {
                int id;
                std::cin >> id;

                node_coord node = find_node(id);

                if (node.line() == network.end()) {
                        std::cerr << "Error: Not found" << std::endl;
                        continue;
                }

                if (!ping(node.ping_sock(), node.ping_port(), id)) {
                        std::cerr << "Error: Node is unavailable" << std::endl;
                        continue;
                }

                std::string line;
                std::getline(std::cin, line);
                std::istringstream is(line);
                std::string name, value;
                is >> name >> value;

                zmqpp::message msg;

                msg << id;
                if (value != "") {
                        msg << static_cast<int>(action::exec_set) << name <<
stoi(value);
                } else {
                        msg << static_cast<int>(action::exec_get) << name;
                }

                node.out_sock().send(msg);
        }

        else if (command == "pingall") {
                std::list<int> unavailable;
                bool dead;

                for (auto& line: network) {
                        zmqpp::socket &ping_node = line.second.second.first;
                        std::string &ping_port = line.second.second.second;
                        dead = false;
                        for (int& node: line.first) {
                                if (dead) {
                                        unavailable.push_back(node);
                                } else if (!ping(ping_node, ping_port, node)) {
                                        dead = true;
                                        unavailable.push_back(node);
                                }
                        }
                }

                std::cout << "OK: ";
                if (unavailable.empty()) {
                        std::cout << -1 << std::endl;
```

```
                } else {
                        std::cout << *unavailable.begin();
                        for (auto node = ++unavailable.begin(); node !=
unavailable.end(); ++node) {
                                std::cout << "; " << *node;
                        }
                        std::cout << std::endl;
                }
        }


        std::cout << "> ";
    }
}

// node.cpp
// Вычислительный узел
#include <unistd.h>
#include <pthread.h>
#include <zmqpp/zmqpp.hpp>
#include <iostream>
#include <string>
#include <memory>
#include <algorithm>
#include <signal.h>
#include <map>

#include "network.hpp"

/*
front_ping-><-\    /-><-back_ping
               \  /
    front_in->\ \/ /<-back_in
               node
   front_out-</    \>-back_out
*/

int id = -1, next_id = -1, prev_id = -1;
std::unique_ptr<zmqpp::socket> front_in(nullptr), front_out(nullptr),
front_ping(nullptr),
                               back_in(nullptr), back_out(nullptr),
back_ping(nullptr);
std::string back_in_port, back_out_port, back_ping_port, front_ping_port;
zmqpp::context context;

void* ping(void*) {
    int packet[2];
    size_t length = 2 * sizeof(int);
    while (*front_ping) {
        front_ping->receive_raw(reinterpret_cast<char *>(packet), length);
        // std::cout << "received ping " << packet[0] << " " << packet[1] <<
std::endl;
        if (packet[0] != id) {
            // std::cout << "sending forward" << std::endl;
            if (back_ping.get() != nullptr) {
                if (!back_ping->send_raw(reinterpret_cast<char
*>(packet), length, zmqpp::socket::dont_wait)) {
                    packet[1] = 0;
                } else if (!back_ping->receive_raw(reinterpret_cast<char
*>(packet), length)) {
                    packet[1] = 0;
                    back_ping->close();
                    back_ping.reset(new zmqpp::socket(context,
zmqpp::socket_type::req));
```

```
                              back_ping-
>set(zmqpp::socket_option::receive_timeout, 1000);
                              back_ping->connect(host + back_ping_port);
                        }
                  } else {
                        front_ping->close();
                        front_ping.reset(new zmqpp::socket(context,
zmqpp::socket_type::rep));
                        front_ping->bind(host + front_ping_port);
                        continue;
                  }
            }
            // std::cout << "sending back" << std::endl;
            front_ping->send_raw(reinterpret_cast<char *>(packet), length);
      }
      return NULL;
}

void* back_to_front(void*) {
      zmqpp::message msg;
      while (back_in.get() == nullptr || *back_in) {
            if (back_in.get() == nullptr) {
                  continue;
            }
            back_in->receive(msg);
            // std::cout << "sending back" << std::endl;
            int rid;
            msg >> rid;
            if (rid != id) {
                  front_out->send(msg);
                  continue;
            }

            int act;
            msg >> act;
            switch (static_cast<action>(act)) {
            case action::rebind_back: {
                  msg >> next_id;
                  back_out->disconnect(host + back_out_port);
                  msg >> back_out_port;
                  back_out->connect(host + back_out_port);

                  back_in->unbind(host + back_in_port);
                  msg >> back_in_port;
                  back_in->bind(host + back_in_port);

                  back_ping->disconnect(host + back_ping_port);
                  msg >> back_ping_port;
                  back_ping->connect(host + back_ping_port);
                  break;
            }

            case action::unbind_back: {
                  back_in.reset(nullptr);
                  back_out.reset(nullptr);
                  back_ping.reset(nullptr);
                  next_id = -1;
                  break;
            }

            default: {}
            }
      }
      return NULL;
```

```
}

int main(int argc, char* argv[]) {
	assert(argc >= 4);
	id = strtoul(argv[1], nullptr, 10);
	prev_id = strtoul(argv[2], nullptr, 10);
	std::string bridge_port(argv[3]);
	// std::cout << getpid() << " " << id << std::endl;

	zmqpp::message msg;

	zmqpp::socket bridge(context, zmqpp::socket_type::req);
	bridge.connect(host + bridge_port);

	front_in.reset(new zmqpp::socket(context, zmqpp::socket_type::pull));
	std::string front_in_port = std::to_string(try_bind(*front_in));

	front_ping.reset(new zmqpp::socket(context, zmqpp::socket_type::rep));
	front_ping_port = std::to_string(try_bind(*front_ping));

	msg << front_in_port << front_ping_port;
	bridge.send(msg);

	bridge.receive(msg);

	std::string front_out_port;
	msg >> front_out_port;
	front_out.reset(new zmqpp::socket(context, zmqpp::socket_type::push));
	front_out->connect(host + front_out_port);

	bridge.disconnect(host + bridge_port);
	bridge.close();
	bridge = zmqpp::socket(context, zmqpp::socket_type::rep);
	bridge_port = std::to_string(try_bind(bridge));

	{
		zmqpp::message ans;
		ans << id << static_cast<int>(action::fork) << getpid();
		front_out->send(ans);
	}

	pthread_t ping_id;
	check(pthread_create(&ping_id, NULL, ping, NULL), -1, "pthread_create
error");
	if (pthread_detach(ping_id) != 0) {
		perror("detach error");
	}

	pthread_t back_to_front_id;
	check(pthread_create(&back_to_front_id, NULL, back_to_front, NULL), -1,
"pthread_create error");
	if (pthread_detach(back_to_front_id) != 0) {
		perror("detach error");
	}

	std::map<std::string, int> dictionary;

	int act;
	int tid;

	while (true) {
		if (!front_in->receive(msg)) {
			perror("");
		}
```

```
            msg >> tid;

            if (tid != id) {
                  // std::cout << "sending forward" << std::endl;
                  back_out->send(msg);
                  continue;
            }

            msg >> act;

            switch (static_cast<action>(act)) {
            case action::fork: {
                  int cid;
                  msg >> cid;

                  if (next_id == -1) {
                        next_id = cid;

                        int pid = fork();
                        check(pid, -1, "fork error");
                        if (pid == 0) {
                              check(execl("node", "node",
std::to_string(cid).c_str(), std::to_string(id).c_str(),
                                                bridge_port.c_str(), NULL), -1,
"execl error");
                        }

                        back_in.reset(new zmqpp::socket(context,
zmqpp::socket_type::pull));
                        back_in_port = std::to_string(try_bind(*back_in));

                        zmqpp::message ports;
                        bridge.receive(ports);

                        ports >> back_out_port >> back_ping_port;
                        ports.pop_back();
                        ports.pop_back();

                        back_out.reset(new zmqpp::socket(context,
zmqpp::socket_type::push));
                        back_ping.reset(new zmqpp::socket(context,
zmqpp::socket_type::req));
                        back_ping->set(zmqpp::socket_option::receive_timeout,
1000);

                        back_out->connect(host + back_out_port);
                        back_ping->connect(host + back_ping_port);

                        ports << back_in_port;
                        bridge.send(ports);
                  }
                  break;
            }

            case action::exec_set: {
                  std::string name;
                  int value;
                  msg >> name >> value;
                  dictionary[name] = value;

                  zmqpp::message ans;
                  ans << id << static_cast<int>(action::exec_set);
                  front_out->send(ans);
```

```
                        break;
                }

                case action::exec_get: {
                        std::string name;
                        msg >> name;

                        zmqpp::message ans;
                        ans << id << static_cast<int>(action::exec_get) << name;
                        auto it = dictionary.find(name);
                        if (it != dictionary.end()) {
                                ans << std::to_string(it->second);
                        } else {
                                ans << "not found";
                        }
                        front_out->send(ans);
                        break;
                }

                case action::exit: {
                        back_in->unbind(host + back_in_port);


                        if (prev_id == -1) {
                                if (next_id != -1) {
                                        zmqpp::message msg2;
                                        msg2 << id <<
static_cast<int>(action::rebind_back) << back_in_port << back_ping_port;
                                        front_out->send(msg2);

                                        zmqpp::message msg3;
                                        msg3 << next_id <<
static_cast<int>(action::rebind_front) << front_out_port;
                                        back_out->send(msg3);
                                } else {
                                        zmqpp::message msg;
                                        msg << id << static_cast<int>(action::exit);
                                        front_out->send(msg);
                                }


                        } else {
                                zmqpp::message msg;
                                msg << id << static_cast<int>(action::exit);
                                front_out->send(msg);

                                if (next_id == -1) {
                                        zmqpp::message msg2;
                                        msg2 << prev_id <<
static_cast<int>(action::unbind_back);
                                        front_out->send(msg2);
                                } else {
                                        zmqpp::message msg2;
                                        msg2 << prev_id <<
static_cast<int>(action::rebind_back) << next_id << back_in_port <<
back_out_port << back_ping_port;
                                        front_out->send(msg2);

                                        zmqpp::message msg3;
                                        msg3 << next_id <<
static_cast<int>(action::rebind_front) << front_out_port;
                                        back_out->send(msg3);
                                }
                        }
```

```
                        return 0;
                }

                case action::rebind_front: {
                        std::string tmp;
                        msg << tmp;
                        front_out->connect(host + tmp);
                        front_out->disconnect(host + front_out_port);
                        front_out_port = tmp;
                }

                default: {}
                }
        }

}

// network.hpp
// Вспомогательные функции
#pragma once

#include <zmqpp/zmqpp.hpp>
#include <string>

#define check(VAL, BADVAL, MSG) if (VAL == BADVAL) { perror(MSG); exit(1); }

const std::string host = "tcp://127.0.0.1:";

int try_bind(zmqpp::socket& socket) {
        static unsigned int port(49152);
        while (true) {
                try {
                // std::cout << "trying " << port;
                        socket.bind(host + std::to_string(port));
                } catch (zmqpp::zmq_internal_exception& ex) {
                        ++port;
                        // std::cout << std::endl;
                        continue;
                }
                // std::cout << " ok" << std::endl;
                return port++;
        }
}

enum class action: int {
        fork, exit, unbind_front, unbind_back, rebind_front, rebind_back,
exec_set, exec_get
};
```

Тесты и протокол исполнения

```
> create 1 -1
OK: 2622
> create 2 -1
OK: 2627
> create 3 2
OK: 2632
> create 4 3
OK: 2637
> create 5 4
```

```
OK: 2705
> pingall
OK: -1
> exec 5 a
OK:5: 'a' not found
> exec 5 a 89
OK:5
> exec 5 a
OK:5: 89
> exec 3 b 45
exec 3 b
exec 2 g 78
exec 1 b
exec 5 a
exec 4 t 90
exec 2 g
>
OK:3
OK:1: 'b' not found
OK:3: 45
OK:2
OK:5: 89
OK:4
OK:2: 78
create 3 5
Error: Already exists
/* kill -9 2632 */
> pingall
OK: 3; 4; 5
> create 6 5
Error: Parent is unavailable
> exec 3 b
Error: Node is unavailable
> exit
```

Выводы

В ходе лабораторной работы я познакомился с очередями сообщений, а также реализовал сеть из вычислительных узлов для асинхронного выполнения вычислений. Очереди сообщений очень хорошо подходят для реализации асинхронного взаимодействия, за исключением того, что невозможно узнать принимает ли кто-то сообщения или нет. Это вызывает затруднения, если важно, какой узел должен выполнить какой запрос. В качестве решения, можно задать время ожидания ответа или постоянно обмениваться с узлами дополнительными сообщениями, подтверждающими, что они работоспособны.

Список литературы

1. ZeroMQ API Reference – URL: http://api.zeromq.org/master:_start
2. ZeroMQ RFC – URL: https://rfc.zeromq.org/

3. Таненбаум Э., Бос Х. *Современные операционные системы. – 4-е изд.* – СПб.: Издательский дом «Питер», 2018.