Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №6 по курсу «Операционные системы»

Студент: А. П. Шорохов
Преподаватель: А. А. Соколов
Группа: М8О-206Б
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №6

Задача: Реализовать распределенную систему по обработке запросов. В данной 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 сервера сообщений zmq. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом.

Вариант задания: 46. **Топология:** дерево общего вида.
Тип вычислительной команды: локальный целочисленный словарь.
Тип проверки узлов на доступность: ping.

1 Описание

Наша программа представляет из себя 2 узла(управляющий и вычислительный), которые получаются при компиляции исходных файлов, а также из статической библиотеки, которая подключается при компиляции к этим файлам. Общение между узлами, как и сказано в задании, реализовано посредством zmq.

2 Алгоритм

1. Управляющий узел принимает команды, обрабатывает их и передаёт дочерним узлам или сообщает об ошибке.
2. Дочерние узлы проверяют возможность выполнения команды в данном узле. Если это невозможно, то команда пересылается в один из дочерних узлов, откуда впоследствии возвращается сообщение об успехе(или неудаче), которое передаётся обратно по дереву.
3. При недоступности узла выводится сообщение, которое затем передается управляющему узлу.
4. При удалении узла все его потомки рекурсивно уничтожаются.

3 Исходный код

sf.h

```
#pragma once
#include <iostream>
#include <string>
#include "zmq.hpp"
#include "unistd.h"

bool send_msg(zmq::socket_t& socket, const std::string& message);

std::string get_msg(zmq::socket_t& socket);

int bind_socket(zmq::socket_t& socket);

void crt_node(int id, int portNumber);
```

sf.cpp

```
#include "sf.h"
bool send_msg(zmq::socket_t& socket, const std::string& message) {
  zmq::message_t m(message.size());
  memcpy(m.data(), message.c_str(), message.size());
  try {
    socket.send(m);
    return true;
  } catch(...) {
    return false;
  }
}

std::string get_msg(zmq::socket_t& socket) {
  zmq::message_t message;
  bool msg_got;
  try {
    msg_got = socket.recv(&message);
  } catch(...) {
    msg_got = false;
  }
  std::string received(static_cast<char*>(message.data()), message.size());
  if(!msg_got || received.empty()) {
    return "Error: Node is unavailable";
  } else {
    return received;
  }
}

int bind_socket(zmq::socket_t& socket) {
  int port = 30000;
```

```
  std::string port_tmp = "tcp://127.0.0.1:";
  while(true) {
    try {
      socket.bind(port_tmp + std::to_string(port));
      break;
    } catch(...) {
      port++;
    }
  }
  return port;
}

void crt_node(int id, int portNumber) {
  char* arg0 = strdup("./child_node");
  char* arg1 = strdup((std::to_string(id)).c_str());
  char* arg2 = strdup((std::to_string(portNumber)).c_str());
  char* args[] = {arg0, arg1, arg2, nullptr};
  execv("./child_node", args);
}
```

childNode.cpp

```
#include <string>
#include <sstream>
#include <zmq.hpp>
#include <csignal>
#include <iostream>
#include <unordered_map>
#include "sf.h"
int main(int argc, char* argv[]) {
  if(argc != 3) {
    std::cerr << "Not enough parameters" << std::endl;
    exit(-1);
  }
  int id = std::stoi(argv[1]);
  int parent_port = std::stoi(argv[2]);
  zmq::context_t ctx;
  zmq::socket_t parent_socket(ctx, ZMQ_REP);
  std::string port_tmp = "tcp://127.0.0.1:";
  parent_socket.connect(port_tmp + std::to_string(parent_port));
  std::unordered_map<int, int> pids;
  std::unordered_map<int, int> ports;
  std::unordered_map<int, zmq::socket_t> sockets;
  while(true) {
    std::string action = get_msg(parent_socket);
    std::stringstream s(action);
    std::string command;
    s >> command;
    if(command == "pid") {
      std::string reply = "Ok: " + std::to_string(getpid());
```

```
29      send_msg(parent_socket, reply);
30    } else if(command == "create") {
31      int size, node_id;
32      s >> size;
33      std::vector<int> path(size);
34      for(int i = 0; i < size; ++i) {
35        s >> path[i];
36      }
37      s >> node_id;
38      if(size == 0) {
39        auto socket = zmq::socket_t(ctx, ZMQ_REQ);
40        socket.setsockopt(ZMQ_SNDTIMEO, 5000);
41        socket.setsockopt(ZMQ_LINGER, 5000);
42        socket.setsockopt(ZMQ_RCVTIMEO, 5000);
43        socket.setsockopt(ZMQ_REQ_CORRELATE, 1);
44        socket.setsockopt(ZMQ_REQ_RELAXED, 1);
45        sockets.emplace(node_id, std::move(socket));
46        int port = bind_socket(sockets.at(node_id));
47        int pid = fork();
48        if(pid == -1) {
49          send_msg(parent_socket, "Unable to fork");
50        } else if(pid == 0) {
51          crt_node(node_id, port);
52        } else {
53          ports[node_id] = port;
54          pids[node_id] = pid;
55          send_msg(sockets.at(node_id), "pid");
56          send_msg(parent_socket, get_msg(sockets.at(node_id)));
57        }
58      } else {
59        int next_smb = path.front();
60        path.erase(path.begin());
61        std::stringstream msg;
62        msg << "create " << path.size();
63        for(int i : path) {
64          msg << " " << i;
65        }
66        msg << " " << node_id;
67        send_msg(sockets.at(next_smb), msg.str());
68        send_msg(parent_socket, get_msg(sockets.at(next_smb)));
69      }
70    } else if(command == "remove") {
71      int size, node_id;
72      s >> size;
73      std::vector<int> path(size);
74      for(int i = 0; i < size; ++i) {
75        s >> path[i];
76      }
77      s >> node_id;
```

```
78      if(path.empty()) {
79        send_msg(sockets.at(node_id), "kill");
80        get_msg(sockets.at(node_id));
81        kill(pids[node_id], SIGTERM);
82        kill(pids[node_id], SIGKILL);
83        pids.erase(node_id);
84        sockets.at(node_id).disconnect(port_tmp + std::to_string(ports[node_id]));
85        ports.erase(node_id);
86        sockets.erase(node_id);
87        send_msg(parent_socket, "Ok");
88      } else {
89        int next_smb = path.front();
90        path.erase(path.begin());
91        std::stringstream msg;
92        msg << "remove " << path.size();
93        for(int i : path) {
94          msg << " " << i;
95        }
96        msg << " " << node_id;
97        send_msg(sockets.at(next_smb), msg.str());
98        send_msg(parent_socket, get_msg(sockets.at(next_smb)));
99      }
100   } else if(command == "exec") {
101     int size;
102     s >> size;
103     std::vector<int> path(size);
104     for(int i = 0; i < size; ++i) {
105       s >> path[i];
106     }
107     if(path.empty()) {
108       send_msg(parent_socket, "Node is available");
109     } else {
110       int next_smb = path.front();
111       path.erase(path.begin());
112       std::stringstream msg;
113       msg << "exec " << path.size();
114       for(int i : path) {
115         msg << " " << i;
116       }
117       std::string received;
118       if(!send_msg(sockets.at(next_smb), msg.str())) {
119         received = "Node is unavailable";
120       } else {
121         received = get_msg(sockets.at(next_smb));
122       }
123       send_msg(parent_socket, received);
124     }
125   } else if(command == "ping") {
126     int size;
```

```
127       s >> size;
128       std::vector<int> path(size);
129       for(int i = 0; i < size; ++i) {
130         s >> path[i];
131       }
132       if(path.empty()) {
133         send_msg(parent_socket, "Ok: 1");
134       } else {
135         int next_smb = path.front();
136         path.erase(path.begin());
137         std::stringstream msg;
138         msg << "ping " << path.size();
139         for(int i : path) {
140           msg << " " << i;
141         }
142         std::string received;
143         if(!send_msg(sockets.at(next_smb), msg.str())) {
144           received = "Node is unavailable";
145         } else {
146           received = get_msg(sockets.at(next_smb));
147         }
148         send_msg(parent_socket, received);
149       }
150     } else if(command == "kill") {
151       for(auto& item : sockets) {
152         send_msg(item.second, "kill");
153         get_msg(item.second);
154         kill(pids[item.first], SIGTERM);
155         kill(pids[item.first], SIGKILL);
156       }
157       send_msg(parent_socket, "Ok");
158     }
159     if(parent_port == 0) {
160       break;
161     }
162   }
163 }
```

main.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <zmq.hpp>
4  #include <vector>
5  #include <csignal>
6  #include <sstream>
7  #include <memory>
8  #include <unordered_map>
9  #include "sf.h"
10
```

```cpp
struct Node {
  Node(int id, std::weak_ptr<Node> parent) : id(id), parent(parent) {};
  int id;
  std::weak_ptr<Node> parent;
  std::unordered_map<int, std::shared_ptr<Node>> children;
  std::unordered_map<std::string, int> dictionary;
};

class General_tree {
  public:
  bool insert(int node_id, int parent_id) {
    if(root == nullptr) {
      root = std::make_shared<Node>(node_id, std::weak_ptr<Node>());
      return true;
    }
    std::vector<int> path = get_path(parent_id);
    if(path.empty()) {
      return false;
    }
    path.erase(path.begin());
    std::shared_ptr<Node> tmp = root;
    for(const auto& node : path) {
      tmp = tmp->children[node];
    }
    tmp->children[node_id] = std::make_shared<Node>(node_id, tmp);
    return true;
  }

  bool rmv(int node_id) {
    std::vector<int> path = get_path(node_id);
    if(path.empty()) {
      return false;
    }
    path.erase(path.begin());
    std::shared_ptr<Node> tmp = root;
    for(const auto& node : path) {
      tmp = tmp->children[node];
    }
    if(tmp->parent.lock()) {
      tmp = tmp->parent.lock();
      tmp->children.erase(node_id);
    } else {
      root = nullptr;
    }
    return true;
  }
  [[nodiscard]] std::vector<int> get_path(int id) const {
    std::vector<int> path;
    if(!get_node(root, id, path)) {
```

```
60        return {};
61      } else {
62        return path;
63      }
64    }
65    void add_dictionary(int id, std::string name, int value) {
66      std::vector<int> path = get_path(id);
67      path.erase(path.begin());
68      std::shared_ptr<Node> tmp = root;
69      for(const auto& node : path) {
70        tmp = tmp->children[node];
71      }
72      tmp->dictionary[name] = value;
73    }
74
75    void find_dictionary(int id, std::string name) {
76      std::vector<int> path = get_path(id);
77      path.erase(path.begin());
78      std::shared_ptr<Node> tmp = root;
79      for(const auto& node : path) {
80        tmp = tmp->children[node];
81      }
82      if (tmp->dictionary.find(name) == tmp->dictionary.end()) {
83        std::cout << "'" << name << "' not found" << std::endl;
84      } else {
85        std::cout << tmp->dictionary[name] << std::endl;
86      }
87    }
88    private:
89    bool get_node(const std::shared_ptr<Node>& current, int id, std::vector<int>& path)
          const {
90      if(!current) {
91        return false;
92      }
93      if(current->id == id) {
94        path.push_back(current->id);
95        return true;
96      }
97      path.push_back(current->id);
98      for(const auto& node : current->children) {
99        if(get_node(node.second, id, path)) {
100         return true;
101       }
102     }
103     path.pop_back();
104     return false;
105   }
106   std::shared_ptr<Node> root = nullptr;
107 };
```

```
108
109 int main() {
110   General_tree tree;
111   std::string command;
112   int child_pid = 0;
113   int child_id = 0;
114   zmq::context_t ctx(1);
115   zmq::socket_t rule_socket(ctx, ZMQ_REQ);
116   rule_socket.setsockopt(ZMQ_SNDTIMEO, 5000);
117   rule_socket.setsockopt(ZMQ_LINGER, 5000);
118   rule_socket.setsockopt(ZMQ_RCVTIMEO, 5000);
119   rule_socket.setsockopt(ZMQ_REQ_CORRELATE, 1);
120   rule_socket.setsockopt(ZMQ_REQ_RELAXED, 1);
121   int port_n = bind_socket(rule_socket);
122   while(std::cin >> command) {
123     if(command == "create") {
124       int node_id, parent_id;
125       std::string result;
126       std::cin >> node_id >> parent_id;
127       if(!child_pid) {
128         child_pid = fork();
129         if(child_pid == -1) {
130           std::cout << "Unable to create process" << std::endl;
131           exit(-1);
132         } else if(child_pid == 0) {
133           crt_node(node_id, port_n);
134         } else {
135           parent_id = 0;
136           child_id = node_id;
137           send_msg(rule_socket, "pid");
138           result = get_msg(rule_socket);
139         }
140       } else {
141         if(!tree.get_path(node_id).empty()) {
142           std::cout << "Error: Already exists" << std::endl;
143           continue;
144         }
145         std::vector<int> path = tree.get_path(parent_id);
146         if(path.empty()) {
147           std::cout << "Error: Parent not found" << std::endl;
148           continue;
149         }
150         path.erase(path.begin());
151         std::stringstream s;
152         s << "create " << path.size();
153         for(int id : path) {
154           s << " " << id;
155         }
156         s << " " << node_id;
```

```
157         send_msg(rule_socket, s.str());
158         result = get_msg(rule_socket);
159       }
160
161       if(result.substr(0, 2) == "Ok") {
162         tree.insert(node_id, parent_id);
163       }
164       std::cout << result << std::endl;
165     } else if(command == "remove") {
166       if(child_pid == 0) {
167         std::cout << "Error: Not found" << std::endl;
168         continue;
169       }
170       int node_id;
171       std::cin >> node_id;
172       if(node_id == child_id) {
173         send_msg(rule_socket, "kill");
174         get_msg(rule_socket);
175         kill(child_pid, SIGTERM);
176         kill(child_pid, SIGKILL);
177         child_id = 0;
178         child_pid = 0;
179         std::cout << "Ok" << std::endl;
180         tree.rmv(node_id);
181         continue;
182       }
183       std::vector<int> path = tree.get_path(node_id);
184       if(path.empty()) {
185         std::cout << "Error: Not found" << std::endl;
186         continue;
187       }
188       path.erase(path.begin());
189       std::stringstream s;
190       s << "remove " << path.size() - 1;
191       for(int i : path) {
192         s << " " << i;
193       }
194       send_msg(rule_socket, s.str());
195       std::string recieved = get_msg(rule_socket);
196       if(recieved.substr(0, 2) == "Ok") {
197         tree.rmv(node_id);
198       }
199       std::cout << recieved << std::endl;
200     } else if(command == "exec") {
201       if(child_pid == 0) {
202         std::cout << "Error: Not found" << std::endl;
203         continue;
204       }
205       int node_id;
```

```
206       std::cin >> node_id;
207       std::string name_value;
208       std::getline(std::cin, name_value);
209       std::vector<int> path = tree.get_path(node_id);
210       if(path.empty()) {
211         std::cout << "Error: Not found" << std::endl;
212         continue;
213       }
214       path.erase(path.begin());
215       std::stringstream s;
216       s << "exec " << path.size();
217       for(int i : path) {
218         s << " " << i;
219       }
220       std::string received;
221       if(!send_msg(rule_socket, s.str())) {
222         received = "Node is unavailable";
223       } else {
224         received = get_msg(rule_socket);
225         if (received == "Node is available") {
226           std::string name;
227           int value;
228           int size_arguments = name_value.size();
229           std::stringstream ss(name_value);
230           bool searchNeeded = true;
231           for (int i = 1; i < size_arguments; ++i) {
232             if (name_value[i] == ' ') {
233               ss >> name;
234               ss >> value;
235               tree.add_dictionary(node_id, name, value);
236               std::cout << "Ok:" << node_id << std::endl;
237               searchNeeded = false;
238               break;
239             }
240           }
241           if (searchNeeded) {
242             ss >> name;
243             std::cout << "Ok:" << node_id << ": ";
244             tree.find_dictionary(node_id, name);
245           }
246         } else {
247           std::cout << received << std::endl;
248         }
249       }
250     } else if(command == "ping") {
251       if(child_pid == 0) {
252         std::cout << "Error: Not found" << std::endl;
253         continue;
254       }
```

```
      int node_id;
      std::cin >> node_id;
      std::vector<int> path = tree.get_path(node_id);
      if(path.empty()) {
        std::cout << "Error: Not found" << std::endl;
        continue;
      }
      path.erase(path.begin());
      std::stringstream s;
      s << "ping " << path.size();
      for(int i : path) {
        s << " " << i;
      }
      std::string received;
      if(!send_msg(rule_socket, s.str())) {
        received = "Node is unavailable";
      } else {
        received = get_msg(rule_socket);
      }
      std::cout << received << std::endl;
    } else if(command == "exit") {
      send_msg(rule_socket, "kill");
      get_msg(rule_socket);
      kill(child_pid, SIGTERM);
      kill(child_pid, SIGKILL);
      break;
    } else {
      std::cout << "Unknown command" << std::endl;
    }
    command.clear();
  }
  return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(os_lab_6)

set(CMAKE_CXX_STANDARD 17)


add_executable(out main.cpp)
add_executable(childNode childNode.cpp)
add_library(functions sf.cpp sf.h)

target_link_libraries(functions zmq)
target_link_libraries(out zmq functions)
target_link_libraries(childNode zmq functions)
```

4 Консоль

```
create 1 -1
Ok: 3354
create 3 1
Ok: 3358
create 5 3
Ok: 3361
create 2 1
Ok: 3364
create 4 1
Ok: 3367
ping 1
Ok: 1
ping 2
Ok: 1
ping 3
Ok: 1
ping 4
Ok: 1
ping 5
Ok: 1
ping 10
Error: Not found
remove 3
Ok
ping 3
Error: Not found
ping 5
Error: Not found
exec 2 hello 777
Ok:2
exec 2 hello
Ok:2: 777
exec 2 hi
Ok:2: 'hi'not found
exit
```

5 Выводы

При написании данной лабораторной работы, я научился работать с очередями сообщений, реализовал собственную распределенную систему по обработке запросов, познакомился с сервером сообщений zmq. Эта лабораторная работа оказалась для меня сложной и на ее написание ушло довольно много времени.