Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №6-8 по курсу «Операционные системы»

Студент: В. М. Ватулин
Преподаватель: А. А. Соколов
Группа: М8О-206Б-19
Дата: 17.04.2021
Оценка:
Подпись:

Москва, 2021

1 Постановка задачи

Цель работы:

Приобретение практических навыков в:

- Управлении серверами сообщений (№6)
- Применение отложенных вычислений (№7)
- Интеграция программных систем друг с другом (№8)

Задание (вариант 19):

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

Удаление существующего вычислительного узла
Формат команды: remove id
id – целочисленный идентификатор удаляемого вычислительного узла Формат вывода:
«Ok» - успешное удаление
«Error: Not found» - вычислительный узел с таким идентификатором не найден
«Error: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом
«Error: [Custom error]» - любая другая обрабатываемая ошибка
Примечание: при удалении узла из топологии его процесс должен быть завершен и работоспособность вычислительной сети не должна быть нарушена.

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
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.

Вариант 19.
Топология 2
Все вычислительные узлы находятся в дереве общего вида. Есть только один управляющий узел.

Набор команд 3 (локальный таймер)
Формат команды сохранения значения: exec id subcommand
subcommand – одна из трех команд: start, stop, time.
start – запустить таймер
stop – остановить таймер
time – показать время локального таймера в миллисекундах
Пример:

```
>exec 10 time
```

```
Ok:10: 0
>exec 10 start
Ok:10
>exec 10 start
Ok:10
*прошло 10 секунд*
>exec 10 time
Ok:10: 10000
*прошло 2 секунды*
>exec 10 stop
Ok:10
*прошло 2 секунды*
>exec 10 time
Ok:10: 12000
```

Команда проверки 3
Формат команды: pingall
Вывод всех недоступных узлов вывести разделенные через точку запятую. Пример:

```
>pingall
Ok: -1 // Все узлы доступны
>pingall
Ok: 7;10;15 // узлы 7,10,15 - недоступны
```

2 Общий метод и алгоритм решения

Для общения между узлами используется очередь сообщений ZeroMQ. Управляющим узлом является программа server, вычислительным - client. client создается с помощью fork() и последующего execl. Библиотека zmq обернута в функции, которые кидают исключение в случае ошибки. Классы Socket, Client и Server созданы для удобства написания кода и инкапсулируют соответствующий их названиям функционал. Сокеты используют протокол ipc, так как он предназначен для межпроцессорного взаимодействия. Используется шаблон Publish-Subscribe, сообщения передаются уровень за уровнем по дереву. Сама структура дерева общего вида описана классом n_tree.

При запуске программы (./server) создается сервер, который разделяется на два потока, один из которых создает нулевой вычислительный узел. Один поток сервера принимает ввод из stdin, другой поток обрабатывает все присланные ему данные. Сообщения по дереву могут передаваться вниз и вверх, клиент, указанный адресатом сообщения выполняет необходимое ему действие и при необходимости отправляет результат обратно в управляющий узел.

3 Исходный код

Заголовочные файлы:

ntree.h

```
1
2  #ifndef _NTREE_H_
3  #define _NTREE_H_
4
5  #include <unistd.h>
6  #include <iostream>
7  #include <memory>
8  #include <unordered_set>
9  #include <utility>
10
11 const std::pair<int, pid_t> BAD_RES = {-1, -1};
12 class n_tree_node {
13 private:
14     std::pair<int, pid_t> id_ = BAD_RES;
15     std::unordered_set<std::shared_ptr<n_tree_node>> children_;
16 public:
17     n_tree_node(std::pair<int, pid_t> id = BAD_RES);
18     bool add_to(int parrent_id, std::pair<int, pid_t> new_id);
19     bool remove(int id);
20     bool find(int id) const;
21     std::pair<int, pid_t>& get(int id);
22     std::unordered_set<pid_t> get_all_second() const;
23     std::unordered_set<int> get_all_first() const;
24     std::pair<int, pid_t> id() const;
25 };
26
27 class n_tree {
28 private:
29     std::shared_ptr<n_tree_node> head_;
30 public:
31     bool add_to(int parrent_id, std::pair<int, pid_t> new_id);
32     bool remove(int id);
33     bool find(int id) const;
34     std::pair<int, pid_t>& get(int id);
35     std::unordered_set<pid_t> get_all_second() const;
36     std::unordered_set<int> get_all_first() const;
37 };
38
39 #endif
```

zmq_wrapper.h

```
1
2  #ifndef _ZMQ_WRAPPER_H_
3  #define _ZMQ_WRAPPER_H_
```

```
4
5  #include <string>
6
7  enum class message_type {
8      ERROR,
9      CHECK,
10     CREATE_CHILD,
11     REMOVE_CHILD,
12     TIMER_TIME,
13     TIMER_START,
14     TIMER_STOP,
15     PINGALL
16 };
17
18 enum class ip_type {
19     CHILD_PUB,
20     PARRENT_PUB,
21 };
22
23 enum class socket_type {
24     PUB,
25     SUB
26 };
27
28
29
30 struct Message {
31     message_type command = message_type::ERROR;
32     int to_id;
33     int value;
34     bool go_up = false;
35     int uniq_num;
36     Message();
37     Message(message_type command_a, int to_id_a, int value_a);
38 };
39
40 void* create_zmq_context();
41 void destroy_zmq_context(void* context);
42 void* create_zmq_socket(void* context, socket_type type);
43 void close_zmq_socket(void* socket);
44 std::string create_ip(ip_type type, pid_t id);
45 void bind_zmq_socket(void* socket, std::string ip);
46 void unbind_zmq_socket(void* socket, std::string ip);
47 void connect_zmq_socket(void* socket, std::string ip);
48 void disconnect_zmq_socket(void* socket, std::string ip);
49 bool operator==(const Message& lhs, const Message& rhs);
50 void send_zmq_msg(void* socket, Message msg);
51 Message get_zmq_msg(void* socket);
52
```

```
53 #endif
```

socket.h

```
1
2  #ifndef _SOCKET_H_
3  #define _SOCKET_H_
4
5  #include <string>
6  #include "zmq_wrapper.h"
7
8  enum class connection_type {
9      BIND,
10     CONNECT
11 };
12
13 class Socket {
14 public:
15     Socket(void* context, socket_type socket_type, std::string ip);
16     ~Socket();
17     void send(Message message);
18     Message receive();
19     void subscribe(std::string ip);
20     std::string ip() const;
21 private:
22     void* socket_;
23     socket_type socket_type_;
24     std::string ip_;
25 };
26
27 #endif
```

server.h

```
1
2  #ifndef _SERVER_H_
3  #define _SERVER_H_
4
5  #include <unistd.h>
6  #include <memory>
7  #include <unordered_map>
8  #include "socket.h"
9  #include "ntree.h"
10
11 class Server {
12 public:
13     Server();
14     ~Server();
15     pid_t pid() const;
16     Message last_message() const;
17     void send(Message message);
```

```
18      Message receive();
19      bool check(int id);
20      void create_child_cmd(int id, int parrent_id);
21      void remove_child_cmd(int id);
22      void exec_cmd(int id, message_type type);
23      void pingall_cmd();
24      friend void* second_thread(void* serv_arg);
25  private:
26      pid_t pid_;
27      void* context_ = nullptr;
28      std::unique_ptr<Socket> pub_;
29      std::unique_ptr<Socket> sub_;
30      pthread_t receive_msg_loop_id;
31      bool terminated_ = false;
32      Message last_message_;
33      n_tree tree_;
34      std::unordered_map<int, bool> map_for_check_;
35  };
36
37  #endif
```

client.h

```
1
2   #ifndef _CLIENT_H_
3   #define _CLIENT_H_
4
5   #include <unistd.h>
6   #include <chrono>
7   #include <memory>
8   #include <string>
9   #include "socket.h"
10
11  class Client {
12  public:
13      Client(int id, std::string parrent_ip);
14      ~Client();
15      int id() const;
16      pid_t pid() const;
17      void send_up(Message message);
18      void send_down(Message message);
19      Message receive();
20      void start_timer();
21      void stop_timer();
22      int get_time();
23      void pingall();
24      void add_child(int id);
25  private:
26      int id_;
27      pid_t pid_;
```

```
28      void* context_ = nullptr;
29      std::unique_ptr<Socket> child_pub_;
30      std::unique_ptr<Socket> parrent_pub_;
31      std::unique_ptr<Socket> sub_;
32      bool is_timer_started = false;
33      std::chrono::steady_clock::time_point start_;
34      std::chrono::steady_clock::time_point finish_;
35      bool terminated_ = false;
36  };
37
38  #endif
```

Файлы с кодом:

ntree.cpp

```
1
2   #include "ntree.h"
3   #include <algorithm>
4   #include <exception>
5
6   n_tree_node::n_tree_node(std::pair<int, pid_t> id) : id_(id) {}
7
8   bool n_tree_node::add_to(int parrent_id, std::pair<int, pid_t> new_id) {
9       if (id_.first == parrent_id) {
10          children_.insert(std::make_shared<n_tree_node>(new_id));
11          return true;
12      }
13      else {
14          bool is_ok = false;
15          for (const auto& ch_ptr : children_) {
16              is_ok = is_ok || ch_ptr->add_to(parrent_id, new_id);
17              if (is_ok) {
18                  break;
19              }
20          }
21          return is_ok;
22      }
23  }
24
25
26
27  bool n_tree_node::remove(int id) {
28
29      auto it = std::find_if(children_.begin(), children_.end(), [id](const auto& ptr) {
30
31                                                                          return
                                                                              ptr
                                                                              ->
                                                                              id_
```

```
                                                                                .
                                                                                first

                                                                                ==

                                                                                id
                                                                                ;
32
33                                                                          });
34
35      if (it != children_.end()) {
36
37          children_.erase(it);
38
39          return true;
40
41      }
42
43      else {
44
45          bool is_ok = false;
46
47          for (const auto& ch_ptr : children_) {
48
49              is_ok = is_ok || ch_ptr->remove(id);
50
51              if (is_ok) {
52
53                  break;
54
55              }
56
57          }
58
59          return is_ok;
60
61      }
62
63  }
64
65
66
67  bool n_tree_node::find(int id) const {
68
69      if (id_.first == id) {
70
71          return true;
72
73      }
```

```
74
75      else {
76
77          bool is_ok = false;
78
79          for (const auto& ch_ptr : children_) {
80
81              is_ok = is_ok || ch_ptr->find(id);
82
83              if (is_ok) {
84
85                  return true;
86
87              }
88
89          }
90
91          return is_ok;
92
93      }
94
95  }
96
97
98
99  std::pair<int, pid_t>& n_tree_node::get(int id) {
100
101     if (id_.first == id) {
102
103         return id_;
104
105     }
106
107     else {
108
109         for (const auto& ch_ptr : children_) {
110
111             if (ch_ptr->find(id)) {
112
113                 return ch_ptr->get(id);
114
115             }
116
117         }
118
119         throw std::runtime_error("can't get value");
120
121     }
122
```

```
123 }
124
125
126
127 std::unordered_set<pid_t> n_tree_node::get_all_second() const {
128
129     std::unordered_set<pid_t> res;
130
131     for (const auto& ptr : children_) {
132
133         res.insert(ptr->id().second);
134
135         res.merge(ptr->get_all_second());
136
137     }
138
139     return res;
140
141 }
142
143
144
145 std::unordered_set<pid_t> n_tree_node::get_all_first() const {
146
147     std::unordered_set<pid_t> res;
148
149     for (const auto& ptr : children_) {
150
151         res.insert(ptr->id().first);
152
153         res.merge(ptr->get_all_first());
154
155     }
156
157     return res;
158
159 }
160
161
162
163 std::pair<int, pid_t> n_tree_node::id() const {
164
165     return id_;
166
167 }
168
169
170
171 bool n_tree::add_to(int parrent_id, std::pair<int, pid_t> new_id) {
```

```
172
173     if (!head_) {
174
175         head_ = std::make_shared<n_tree_node>(new_id);
176
177         return true;
178
179     }
180
181     if (find(new_id.first)) {
182
183         return false;
184
185     }
186
187     return head_->add_to(parrent_id, new_id);
188
189 }
190
191
192
193 bool n_tree::remove(int id) {
194
195     if (head_) {
196
197         if (id == head_->id().first) {
198
199             head_ = nullptr;
200
201             return true;
202
203         }
204
205         return head_->remove(id);
206
207     }
208
209     return false;
210
211 }
212
213
214
215 bool n_tree::find(int id) const {
216
217     if (head_) {
218
219         return head_->find(id);
220
```

```
221     }
222
223     return false;
224
225 }
226
227
228
229 std::pair<int, pid_t>& n_tree::get(int id) {
230
231     if (head_) {
232
233         return head_->get(id);
234
235     }
236
237     throw std::runtime_error("can't get value");
238
239 }
240
241
242
243 std::unordered_set<pid_t> n_tree::get_all_second() const {
244
245     if (head_) {
246
247         std::unordered_set<pid_t> res;
248
249         res.insert(head_->id().second);
250
251         res.merge(head_->get_all_second());
252
253         return res;
254
255     }
256
257     return {};
258
259 }
260
261
262
263 std::unordered_set<pid_t> n_tree::get_all_first() const {
264
265     if (head_) {
266
267         std::unordered_set<pid_t> res;
268
269         res.insert(head_->id().first);
```

```
270
271         res.merge(head_->get_all_first());
272
273         return res;
274
275     }
276
277     return {};
278
279 }
```

zmq_wrapper.cpp

```
1
2  #include "zmq_wrapper.h"
3
4
5
6  #include <errno.h>
7
8  #include <string.h>
9
10 #include <unistd.h>
11
12 #include <zmq.h>
13
14
15
16 #include <iostream>
17
18 #include <tuple>
19
20
21
22
23
24 void* create_zmq_context() {
25
26     void* context = zmq_ctx_new();
27
28     if (context == NULL) {
29
30         throw std::runtime_error(zmq_strerror(errno));
31
32     }
33
34     return context;
35
36 }
37
```

```
38
39
40  void destroy_zmq_context(void* context) {
41
42    if (zmq_ctx_destroy(context) != 0) {
43
44      throw std::runtime_error(zmq_strerror(errno));
45
46    }
47
48  }
49
50
51
52  int get_zmq_socket_type(socket_type type) {
53
54      switch (type) {
55
56          case socket_type::PUB:
57
58              return ZMQ_PUB;
59
60          case socket_type::SUB:
61
62              return ZMQ_SUB;
63
64          default:
65
66              throw std::logic_error("Undefined socket type");
67
68      }
69
70  }
71
72
73
74  void* create_zmq_socket(void* context, socket_type type) {
75
76      int zmq_type = get_zmq_socket_type(type);
77
78      void* socket = zmq_socket(context, zmq_type);
79
80      if (socket == NULL) {
81
82          throw std::runtime_error(zmq_strerror(errno));
83
84      }
85
86      if (zmq_type == ZMQ_SUB) {
```

```
87
88          if (zmq_setsockopt(socket, ZMQ_SUBSCRIBE, 0, 0) == -1) {
89
90              throw std::runtime_error(zmq_strerror(errno));
91
92          }
93
94          int linger_period = 0;
95
96          if (zmq_setsockopt(socket, ZMQ_LINGER, &linger_period, sizeof(int)) == -1) {
97
98              throw std::runtime_error(zmq_strerror(errno));
99
100         }
101
102     }
103
104     return socket;
105
106 }
107
108
109
110 void close_zmq_socket(void* socket) {
111
112     sleep(1);
113
114     if (zmq_close(socket) != 0) {
115
116         throw std::runtime_error(zmq_strerror(errno));
117
118     }
119
120 }
121
122
123
124 std::string create_ip(ip_type type, pid_t id) {
125
126     switch (type) {
127
128         case ip_type::PARRENT_PUB:
129
130             return "ipc:///tmp/parrent_pub_" + std::to_string(id);
131
132         case ip_type::CHILD_PUB:
133
134             return "ipc:///tmp/child_pub_" + std::to_string(id);
135
```

```
136         default:
137 
138             throw std::logic_error("undefined ip type");
139 
140     }
141 
142 }
143 
144 
145 
146 void bind_zmq_socket(void* socket, std::string ip) {
147 
148     if (zmq_bind(socket, ip.data()) != 0) {
149 
150         throw std::runtime_error(zmq_strerror(errno));
151 
152     }
153 
154 }
155 
156 
157 
158 void unbind_zmq_socket(void* socket, std::string ip) {
159 
160     if (zmq_unbind(socket, ip.data()) != 0) {
161 
162         throw std::runtime_error(zmq_strerror(errno));
163 
164     }
165 
166 }
167 
168 
169 
170 void connect_zmq_socket(void* socket, std::string ip) {
171 
172     if (zmq_connect(socket, ip.data()) != 0) {
173 
174         throw std::runtime_error(zmq_strerror(errno));
175 
176     }
177 
178 }
179 
180 
181 
182 void disconnect_zmq_socket(void* socket, std::string ip) {
183 
184     if (zmq_disconnect(socket, ip.data()) != 0) {
```

```
185
186         throw std::runtime_error(zmq_strerror(errno));
187
188     }
189
190 }
191
192
193
194 int counter = 0;
195
196 Message::Message() {
197
198     uniq_num = counter++;
199
200 }
201
202
203
204 Message::Message(message_type command_a, int to_id_a, int value_a)
205
206     : Message() {
207
208     command = command_a;
209
210     to_id = to_id_a;
211
212     value = value_a;
213
214 }
215
216
217
218 bool operator==(const Message& lhs, const Message& rhs) {
219
220     return std::tie(lhs.command, lhs.to_id, lhs.value, lhs.uniq_num) == std::tie(rhs.
            command, rhs.to_id, rhs.value, rhs.uniq_num);
221
222 }
223
224
225
226 void create_zmq_msg(zmq_msg_t* zmq_msg, Message msg) {
227
228     zmq_msg_init_size(zmq_msg, sizeof(Message));
229
230     memcpy(zmq_msg_data(zmq_msg), &msg, sizeof(Message));
231
232 }
```

```
233
234
235
236 void send_zmq_msg(void* socket, Message msg) {
237
238     zmq_msg_t zmq_msg;
239
240     create_zmq_msg(&zmq_msg, msg);
241
242     if (!zmq_msg_send(&zmq_msg, socket, 0)) {
243
244         throw std::runtime_error(zmq_strerror(errno));
245
246     }
247
248     zmq_msg_close(&zmq_msg);
249
250 }
251
252
253
254 Message get_zmq_msg(void* socket) {
255
256     zmq_msg_t zmq_msg;
257
258     zmq_msg_init(&zmq_msg);
259
260     if (zmq_msg_recv(&zmq_msg, socket, 0) == -1) {
261
262         return Message{message_type::ERROR, 0, 0};
263
264     }
265
266     Message msg;
267
268     memcpy(&msg, zmq_msg_data(&zmq_msg), sizeof(Message));
269
270     zmq_msg_close(&zmq_msg);
271
272     return msg;
273
274 }
```

socket.cpp

```
1
2 #include "socket.h"
3
4
5
```

```cpp
#include <iostream>

#include <stdexcept>




Socket::Socket(void* context, socket_type socket_type, std::string ip)

    : socket_type_(socket_type), ip_(ip) {

    socket_ = create_zmq_socket(context, socket_type_);

    switch (socket_type_) {

        case socket_type::PUB:

            bind_zmq_socket(socket_, ip);

            break;

        case socket_type::SUB:

            connect_zmq_socket(socket_, ip);

            break;

        default:

            throw std::logic_error("Undefined connection type");

    }

}



Socket::~Socket() {

    try {

        close_zmq_socket(socket_);

    }

    catch (std::exception& ex) {

        std::cerr << "Socket wasn't closed: " << ex.what() << std::endl;
```

```
55
56      }
57
58  }
59
60
61
62  void Socket::send(Message message) {
63
64      if (socket_type_ == socket_type::PUB) {
65
66          send_zmq_msg(socket_, message);
67
68      }
69
70      else {
71
72          throw std::logic_error("SUB socket can't send messages");
73
74      }
75
76  }
77
78
79
80  Message Socket::receive() {
81
82      if (socket_type_ == socket_type::SUB) {
83
84          return get_zmq_msg(socket_);
85
86      }
87
88      else {
89
90          throw std::logic_error("PUB socket can't receive messages");
91
92      }
93
94  }
95
96
97
98  void Socket::subscribe(std::string ip) {
99
100     if (socket_type_ == socket_type::SUB) {
101
102         connect_zmq_socket(socket_, ip);
103
```

```
104     }
105
106     else {
107
108         throw std::logic_error("Subscribe is only for SUB sockets");
109
110     }
111
112 }
113
114
115
116 std::string Socket::ip() const {
117
118     return ip_;
119
120 }
```

server.cpp

```
1
2  #include "server.h"
3
4
5
6  #include <pthread.h>
7
8  #include <signal.h>
9
10 #include <unistd.h>
11
12 #include <cstring>
13
14 #include <cerrno>
15
16
17
18 #include <iostream>
19
20
21
22 #include "zmq_wrapper.h"
23
24
25
26
27
28 const int ERR_LOOP = 2;
29
30 const std::string CLIENT_PROG = "./client";
```

```
31
32  const double MESSAGE_WAITING_TIME = 1;
33
34  const int MESSAGE_ALL = -256;
35
36
37
38  void* second_thread(void* serv_arg) {
39
40      Server* server_ptr = (Server*)serv_arg;
41
42      pid_t server_pid = server_ptr->pid();
43
44      try {
45
46          pid_t child_pid = fork();
47
48          if (child_pid == -1) {
49
50              throw std::runtime_error(std::strerror(errno));
51
52          }
53
54          if (child_pid == 0) {
55
56              execl(CLIENT_PROG.data(), CLIENT_PROG.data(), "0", server_ptr->pub_->ip().
                    data(), NULL);
57
58          }
59
60
61
62          std::string ip = create_ip(ip_type::PARRENT_PUB, child_pid);
63
64          server_ptr->sub_ = std::make_unique<Socket>(server_ptr->context_, socket_type::
                SUB, ip);
65
66          server_ptr->tree_.add_to(0, {0, child_pid});
67
68
69
70          while (true) {
71
72              Message msg = server_ptr->sub_->receive();
73
74              if (msg.command == message_type::ERROR) {
75
76                  if (server_ptr->terminated_) {
77
```

```
78                  return NULL;
79
80              }
81
82              else {
83
84                  throw std::runtime_error("error message");
85
86              }
87
88          }
89
90          server_ptr->last_message_ = msg;
91
92
93
94          switch (msg.command) {
95
96              case message_type::CREATE_CHILD: {
97
98                  auto& pa = server_ptr->tree_.get(msg.to_id);
99
100                 pa.second = msg.value;
101
102                 std::cout << "OK: " << server_ptr->last_message_.value << std::endl;
103
104                 break;
105
106             }
107
108             case message_type::REMOVE_CHILD:
109
110                 server_ptr->tree_.remove(msg.to_id);
111
112                 std::cout << "OK" << std::endl;
113
114                 break;
115
116             case message_type::TIMER_START:
117
118                 std::cout << "OK:" << msg.to_id << std::endl;
119
120                 break;
121
122             case message_type::TIMER_STOP:
123
124                 std::cout << "OK:" << msg.to_id << std::endl;
125
126                 break;
```

```
127 |
128 |                 case message_type::TIMER_TIME: {
129 |
130 |                     std::cout << "OK:" << msg.to_id << ": " << msg.value << std::endl;
131 |
132 |                     break;
133 |
134 |                 }
135 |
136 |             case message_type::PINGALL:
137 |
138 |                 server_ptr->map_for_check_[msg.to_id] = true;
139 |
140 |                 break;
141 |
142 |             default:
143 |
144 |                 break;
145 |
146 |           }
147 |
148 |       }
149 |
150 |   }
151 |
152 |   catch (std::exception& ex) {
153 |
154 |       std::cerr << ex.what() << std::endl;
155 |
156 |       exit(ERR_LOOP);
157 |
158 |   }
159 |
160 |   return NULL;
161 |
162 | }
163 |
164 |
165 |
166 | Server::Server() {
167 |
168 |   pid_ = getpid();
169 |
170 |   context_ = create_zmq_context();
171 |
172 |   std::string ip = create_ip(ip_type::CHILD_PUB, getpid());
173 |
174 |   pub_ = std::make_unique<Socket>(context_, socket_type::PUB, ip);
175 |
```

```
176     if (pthread_create(&receive_msg_loop_id, 0, second_thread, this) != 0) {
177 
178         throw std::runtime_error(std::strerror(errno));
179 
180     }
181 
182 }
183 
184 
185 
186 Server::~Server() {
187 
188     if (terminated_) {
189 
190         std::cerr << std::to_string(pid_) + " server double termination" << std::endl;
191 
192         return;
193 
194     }
195 
196     terminated_ = true;
197 
198     for (pid_t pid : tree_.get_all_second()) {
199 
200         kill(pid, SIGINT);
201 
202     }
203 
204 
205 
206     try {
207 
208         pub_ = nullptr;
209 
210         sub_ = nullptr;
211 
212         destroy_zmq_context(context_);
213 
214     }
215 
216     catch (std::exception& ex) {
217 
218         std::cerr << ex.what() << std::endl;
219 
220     }
221 
222 }
223 
224 
```

```
225
226 void Server::send(Message message) {
227
228     message.go_up = false;
229
230     pub_->send(message);
231
232 }
233
234
235
236 Message Server::receive() {
237
238     return sub_->receive();
239
240 }
241
242
243
244 pid_t Server::pid() const {
245
246     return pid_;
247
248 }
249
250
251
252 Message Server::last_message() const {
253
254     return last_message_;
255
256 }
257
258
259
260 bool Server::check(int id) {
261
262     Message msg(message_type::CHECK, id, 0);
263
264     send(msg);
265
266     sleep(MESSAGE_WAITING_TIME);
267
268     if (last_message_ == msg) {
269
270         return true;
271
272     } else {
273
```

```
274         return false;
275
276     }
277
278 }
279
280
281
282 void Server::create_child_cmd(int id, int parrent_id) {
283
284     if (tree_.find(id)) {
285
286         std::cout << "Error: Already exists" << std::endl;
287
288         return;
289
290     }
291
292     if (!tree_.find(parrent_id)) {
293
294         std::cout << "Error: Parent not found" << std::endl;
295
296         return;
297
298     }
299
300     if (!check(parrent_id)) {
301
302         std::cout << "Error: Parent is unavailable" << std::endl;
303
304         return;
305
306     }
307
308     send(Message(message_type::CREATE_CHILD, parrent_id, id));
309
310     tree_.add_to(parrent_id, {id, 0});
311
312 }
313
314
315
316 void Server::remove_child_cmd(int id) {
317
318     if (id == 0) {
319
320         std::cout << "Error: Can't remove server node" << std::endl;
321
322         return;
```

```
323
324     }
325
326     if (!tree_.find(id)) {
327
328         std::cout << "Error: Not found" << std::endl;
329
330         return;
331
332     }
333
334     if (!check(id)) {
335
336         std::cout << "Error: Node is unavailable" << std::endl;
337
338         return;
339
340     }
341
342     send(Message(message_type::REMOVE_CHILD, id, 0));
343
344 }
345
346
347
348 void Server::exec_cmd(int id, message_type type) {
349
350     if (!tree_.find(id)) {
351
352         std::cout << "Error: Not found" << std::endl;
353
354         return;
355
356     }
357
358     if (!check(id)) {
359
360         std::cout << "Error: Node is unavailable" << std::endl;
361
362         return;
363
364     }
365
366     send(Message(type, id, 0));
367
368 }
369
370
371
```

```
372 void Server::pingall_cmd() {
373
374     send(Message(message_type::PINGALL, MESSAGE_ALL, 0));
375
376     auto uset = tree_.get_all_first();
377
378     for (int id : uset) {
379
380         map_for_check_[id] = false;
381
382     }
383
384     sleep(MESSAGE_WAITING_TIME);
385
386     std::cout << "Ok: ";
387
388     for (auto& [id, bit] : map_for_check_) {
389
390         if (!bit) {
391
392             std::cout << id << ";";
393
394         }
395
396         bit = false;
397
398     }
399
400     std::cout << std::endl;
401
402 }
```

client.cpp

```
1
2  #include "client.h"
3
4
5
6  #include <unistd.h>
7
8
9
10 #include <iostream>
11
12
13
14 #include "zmq_wrapper.h"
15
16
```

```
17
18  //using namespace std;
19
20
21
22  const std::string CLIENT_PROG = "./client";
23
24  const double MESSAGE_WAITING_TIME = 1;
25
26
27
28  Client::Client(int id, std::string parrent_ip) {
29
30      id_ = id;
31
32      pid_ = getpid();
33
34      context_ = create_zmq_context();
35
36
37
38      std::string ip = create_ip(ip_type::CHILD_PUB, getpid());
39
40      child_pub_ = std::make_unique<Socket>(context_, socket_type::PUB, ip);
41
42      ip = create_ip(ip_type::PARRENT_PUB, getpid());
43
44      parrent_pub_ = std::make_unique<Socket>(context_, socket_type::PUB, ip);
45
46
47
48      sub_ = std::make_unique<Socket>(context_, socket_type::SUB, parrent_ip);
49
50
51
52      sleep(MESSAGE_WAITING_TIME);
53
54      send_up(Message(message_type::CREATE_CHILD, id, getpid()));
55
56  }
57
58
59
60  Client::~Client() {
61
62      if (terminated_) {
63
64          std::cerr << std::to_string(pid_) << " Client double termination" << std::endl;
65
```

```
66          return;
67
68      }
69
70
71
72      terminated_ = true;
73
74
75
76      sleep(MESSAGE_WAITING_TIME);
77
78      try {
79
80          child_pub_ = nullptr;
81
82          parrent_pub_ = nullptr;
83
84          sub_ = nullptr;
85
86          destroy_zmq_context(context_);
87
88      }
89
90      catch (std::exception& ex) {
91
92          std::cerr << std::to_string(pid_) << ex.what() << std::endl;
93
94      }
95
96  }
97
98
99
100 void Client::send_up(Message message) {
101
102     message.go_up = true;
103
104     parrent_pub_->send(message);
105
106 }
107
108
109
110 void Client::send_down(Message message) {
111
112     message.go_up = false;
113
114     child_pub_->send(message);
```

```
115
116 }
117
118
119
120 Message Client::receive() {
121
122     Message msg = sub_->receive();
123
124     return msg;
125
126 }
127
128
129
130 void Client::add_child(int id) {
131
132     pid_t child_pid = fork();
133
134     if (child_pid == -1) {
135
136         throw std::runtime_error("Can't fork");
137
138     }
139
140     if (child_pid == 0) {
141
142         execl(CLIENT_PROG.data(), CLIENT_PROG.data(), std::to_string(id).data(),
                child_pub_->ip().data(), NULL);
143
144     }
145
146
147
148     std::string ip = create_ip(ip_type::PARRENT_PUB, child_pid);
149
150     sub_->subscribe(ip);
151
152 }
153
154
155
156 void Client::start_timer() {
157
158     is_timer_started = true;
159
160     start_ = std::chrono::steady_clock::now();
161
162 }
```

```
163
164
165
166 void Client::stop_timer() {
167
168     is_timer_started = false;
169
170     finish_ = std::chrono::steady_clock::now();
171
172 }
173
174
175
176 int Client::get_time() {
177
178     if (is_timer_started) {
179
180       finish_ = std::chrono::steady_clock::now();
181
182     }
183
184     return std::chrono::duration_cast<std::chrono::milliseconds>(finish_ - start_).
            count();
185
186 }
187
188
189
190 void Client::pingall() {
191
192     send_up(Message(message_type::PINGALL, id_, 0));
193
194 }
195
196
197
198 int Client::id() const {
199
200     return id_;
201
202 }
203
204
205
206 pid_t Client::pid() const {
207
208     return pid_;
209
210 }
```

server_main.cpp

```
1
2  #include <signal.h>
3
4
5
6  #include <iostream>
7
8  #include <string>
9
10 #include <cstring>
11
12 #include <cerrno>
13
14
15
16 #include "server.h"
17
18
19
20
21
22 Server* server_ptr = nullptr;
23
24 void terminate(int) {
25
26     if (server_ptr != nullptr) {
27
28         server_ptr->~Server();
29
30     }
31
32     exit(0);
33
34 }
35
36
37
38 void process_cmd(Server& server, std::string cmd) {
39
40     if (cmd == "create") {
41
42         int id, parrent_id;
43
44         std::cin >> id >> parrent_id;
45
46         server.create_child_cmd(id, parrent_id);
47
48     }
```

```
49
50      else if (cmd == "remove") {
51
52          int id;
53
54          std::cin >> id;
55
56          server.remove_child_cmd(id);
57
58      }
59
60      else if (cmd == "exec") {
61
62          int id;
63
64          std::string sub_cmd;
65
66          std::cin >> id >> sub_cmd;
67
68          message_type type;
69
70          if (sub_cmd == "time") {
71
72              type = message_type::TIMER_TIME;
73
74          }
75
76          else if (sub_cmd == "start") {
77
78              type = message_type::TIMER_START;
79
80          }
81
82          else if (sub_cmd == "stop") {
83
84              type = message_type::TIMER_STOP;
85
86          }
87
88          else {
89
90              std::cout << "incorrect arguments" << std::endl;
91
92              return;
93
94          }
95
96          server.exec_cmd(id, type);
97
```

```
98      }
99
100     else if (cmd == "pingall") {
101
102         server.pingall_cmd();
103
104     }
105
106     else {
107
108         std::cout << "no such command" << std::endl;
109
110     }
111
112 }
113
114
115
116 int main() {
117
118     try {
119
120         signal(SIGINT, terminate);
121
122         signal(SIGSEGV, terminate);
123
124         Server server;
125
126         server_ptr = &server;
127
128         std::string cmd;
129
130         while (std::cin >> cmd) {
131
132             process_cmd(server, cmd);
133
134         }
135
136     }
137
138     catch (std::exception& ex) {
139
140         std::cerr << std::to_string(getpid()) << ' ' << ex.what() << std::endl;
141
142         exit(1);
143
144     }
145
146     return 0;
```

```
147 
148 }
```

client_main.cpp

```
1  
2  #include <signal.h>
3  
4  
5  
6  #include <iostream>
7  
8  #include <string>
9  
10 #include <cerrno>
11 
12 #include <cstring>
13 
14 
15 
16 #include "client.h"
17 
18 
19 
20 
21 
22 const int ERR_TERMINATED = 1;
23 
24 const int MESSAGE_ALL = -256;
25 
26 
27 
28 Client* client_ptr = nullptr;
29 
30 
31 
32 void terminate(int) {
33 
34     if (client_ptr != nullptr) {
35 
36         client_ptr->~Client();
37 
38     }
39 
40     exit(0);
41 
42 }
43 
44 
45 
```

```
46 void process_msg(Client& client, const Message msg) {
47
48     switch (msg.command) {
49
50         case message_type::ERROR:
51
52             throw std::runtime_error("error message received");
53
54         case message_type::CHECK:
55
56             client.send_up(msg);
57
58             break;
59
60         case message_type::CREATE_CHILD:
61
62             client.add_child(msg.value);
63
64             break;
65
66         case message_type::REMOVE_CHILD: {
67
68             if (msg.to_id != MESSAGE_ALL) {
69
70                 client.send_up(msg);
71
72             }
73
74             Message tmp = msg;
75
76             tmp.to_id = MESSAGE_ALL;
77
78             client.send_down(tmp);
79
80             terminate(0);
81
82             break;
83
84         }
85
86         case message_type::TIMER_START:
87
88             client.start_timer();
89
90             client.send_up(msg);
91
92             break;
93
94         case message_type::TIMER_STOP:
```

```
95
96              client.stop_timer();
97
98              client.send_up(msg);
99
100             break;
101
102         case message_type::TIMER_TIME: {
103
104             int val = client.get_time();
105
106             client.send_up(Message(message_type::TIMER_TIME, client.id(), val));
107
108             break;
109
110         }
111
112         case message_type::PINGALL:
113
114             client.send_down(msg);
115
116             client.pingall();
117
118             break;
119
120         default:
121
122             throw std::logic_error("no such message command");
123
124     }
125
126 }
127
128
129
130 int main(int argc, char const* argv[]) {
131
132     try {
133
134         signal(SIGINT, terminate);
135
136         signal(SIGSEGV, terminate);
137
138         Client client(std::stoi(argv[1]), std::string(argv[2]));
139
140         client_ptr = &client;
141
142         while (true) {
143
```

```
144             Message msg = client.receive();
145 
146             if (msg.to_id != client.id() && msg.to_id != MESSAGE_ALL) {
147 
148                 if (msg.go_up) {
149 
150                     client.send_up(msg);
151 
152                 }
153 
154                 else {
155 
156                     client.send_down(msg);
157 
158                 }
159 
160                 continue;
161 
162             }
163 
164             process_msg(client, msg);
165 
166         }
167 
168     }
169 
170     catch (std::exception& ex) {
171 
172         std::cerr << std::to_string(getpid()) << ' ' << ex.what() << std::endl;
173 
174         exit(ERR_TERMINATED);
175 
176     }
177 
178     return 0;
179 
180 }
```

4 Пример работы

Продемонстрирую процесс работы программы:

```
eri412@Eri-PC:~/Desktop/study/OS/OSlab6/src$ ./server
OK: 34120
create 1 0
OK: 34131
create 2 1
OK: 34134
create 3 1
OK: 34137
create 4 2
OK: 34140
create 5 3
OK: 34143
exec 5 start
OK:5
exec 5 time
OK:5: 10737
exec 5 stop
OK:5
exec 5 time
OK:5: 24319
exec 5 time
OK:5: 24319
remove 3
OK
exec 5 time
Error: Not found
pingall
Ok:
^Z
[1]+  Stopped                 ./server
eri412@Eri-PC:~/Desktop/study/OS/OSlab6/src$ ps
PID TTY          TIME CMD
34059 pts/0    00:00:00 bash
34116 pts/0    00:00:00 server
34120 pts/0    00:00:00 client
34131 pts/0    00:00:00 client
```

```
34134 pts/0    00:00:00 client
34137 pts/0    00:00:00 client <defunct>
34140 pts/0    00:00:00 client
34147 pts/0    00:00:00 ps
eri412@Eri-PC:~/Desktop/study/OS/OSlab6/src$ kill -9 34134
eri412@Eri-PC:~/Desktop/study/OS/OSlab6/src$ fg
./server
pingall
Ok: 2;4;
exec 2 start
Error: Node is unavailable
eri412@Eri-PC:~/Desktop/study/OS/OSlab6/src$ ps
PID TTY          TIME CMD
34059 pts/0    00:00:00 bash
34149 pts/0    00:00:00 ps
```

5 Вывод

При написании 6-8 лабораторной работы я ознакомился с принципом действия ZeroMQ и очередей сообщений, хоть и не познал данннную технологию в полной мере. Использовать ее оказалось намного удобнее, чем стандартные сокеты, на которых она построена. Однако же сама оригинальная документация библиотеки оказалась не слишком удобна, как мне показалось, она рассчитана на человека, который уже имел дело с данной сферой IT. Лабораторную пришлось писать на C++ в главную очередь из-за наличия STL и наличия умных указателей и деструкторов в частности. Это сильно облегчило работу с памятью. В ходе написания лабораторной работы я улучшил свой навык разделения задачи на подзадачи, а также навык разбиения на простые для понимания абстракции.