МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Институт №8 "Информационные технологий и прикладная математика"
Кафедра 806 "Вычислительная математика и программирование"

Лабораторная работа №6
по курсу "Операционные системы"
3 семестр

Студент: Леухин М. В.
Группа: М8О-206Б-20
Преподаватель: Соколов А. А.
Дата: 29.12.21
Оценка: 4
Подпись: _____

Москва, 2021

Содержание

1 Постановка задачи

Цель работы: приобретение практических навыков в управлении серверами сообщений, применении отложенных вычислений и интеграции программных систем друг с другом.

Задание: реализовать распределённую систему по асинхронной обработке запросов. В данной системе должно существовать два вида узлов: "управдяющий"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:

- create id — создание нового вычислительного узла
- exec id params — исполнение команды на вычислительном узле
- heartbeat time — проверка доступности узлов

Вариант 29: бинарное дерево поиска, сумма чисел, heartbeat.

2 Основная часть

2.1 Листинг программы

Файл server.cpp:

```
#include <iostream>
#include <vector>
#include <unistd.h>
#include <csignal>
#include "headers/message.h"
#include "headers/socket.h"
#include "headers/tree.h"

#define SECOND 1'000'000

void *receiveFunction(void *server);

void *heartbeatFunction(void *server);

class Server {
public:

    void commandProcessing(const string &cmd) {
        if (cmd == "create") {
            int id;
            cin >> id;
            createChild(id);
        } else if (cmd == "exec") {
            int id;
            cin >> id;
            int n;
            cin >> n;
            execChild(id, n);
        } else if (cmd == "exit") {
            throw invalid_argument("Exiting...");
        } else if (cmd == "heartbeat") {
            heartbeat();
        } else if (cmd == "status") {
            int id;
            cin >> id;
            if (!getTree().find(id)) {
                throw runtime_error("Error: node " +
                    to_string(id) + "  doesn't exist");
            }
            if (check(id)) {
                cout << "OK" << endl;
            } else {
                cout << "Node " + to_string(id) + " is
```

```
                    unavailable" << endl;
43              }
44          } else {
45              cout << "invalid command\n";
46          }
47      }
48
49      Server() {
50          context = createContext();
51          pid = getpid();
52          string address =
                createAddress(AddressType::CHILD_PUB_LEFT, pid);
53          publisher = new Socket(context, SocketType::PUBLISHER,
                address);
54          if (pthread_create(&receiveMessage, nullptr,
                receiveFunction, this) != 0) {
55              throw runtime_error("thread create error");
56          }
57          working = true;
58      }
59
60      ~Server() {
61          if (!working) return;
62          working = false;
63          send(Message(CommandType::REMOVE_CHILD, 0, 0));
64          try {
65              delete publisher;
66              delete subscriber;
67              publisher = nullptr;
68              subscriber = nullptr;
69              destroyContext(context);
70              usleep(7.5 * SECOND);
71          } catch (runtime_error &err) {
72              cout << "Server wasn't stopped " << err.what() <<
                    endl;
73          }
74      }
75
76      void send(Message msg) {
77          msg.withoutProcessing = false;
78          publisher->send(msg);
79      }
80
81      void createChild(int id) {
82          if (t.find(id)) {
83              throw runtime_error("Error: node " + to_string(id) +
                    " already exists");
84          }
85          if (t.getPlace(id) && !check(t.getPlace(id))) {
86              throw runtime_error("Error: parent node " +
```

```
                to_string(t.getPlace(id)) + " is unavailable");
87          }
88          send(Message(CommandType::CREATE_CHILD, t.getPlace(id),
                id));
89          t.insert(id);
90      }
91
92      void execChild(int id, int n) {
93          double nums[n];
94          for (int i = 0; i < n; ++i) {
95              int cur;
96              cin >> cur;
97              nums[i] = cur;
98          }
99          if (!t.find(id)) {
100             throw runtime_error("Error: node " + to_string(id) +
                    " doesn't exist");
101         }
102         if (!check(id)) {
103             throw runtime_error("Error: node " + to_string(id) +
                    " is unavailable");
104         }
105         send(Message(CommandType::EXEC_CHILD, id, n, nums, 0));
106     }
107
108     bool check(int id) {
109         Message msg(CommandType::RETURN, id, 0);
110         send(msg);
111         usleep(SECOND);
112         msg.getToIndex() = SERVER_ID;
113         return lastMessage == msg;
114     }
115
116     bool check(int id, int time) {
117         Message msg(CommandType::RETURN, id, 0);
118         send(msg);
119         usleep(4000 * time);
120         msg.getToIndex() = SERVER_ID;
121         return lastMessage == msg;
122     }
123
124     Socket *&getPublisher() {
125         return publisher;
126     }
127
128     Socket *&getSubscriber() {
129         return subscriber;
130     }
131
132     void *getContext() {
```

```
133         return context;
134     }
135
136     Tree &getTree() {
137         return t;
138     }
139
140     Message lastMessage;
141
142     pthread_t heartbeatThread;
143     int heartbeatTime;
144     bool isHeartbeat;
145
146     void heartbeat() {
147         if (!isHeartbeat) {
148             int time;
149             cin >> time;
150             heartbeatTime = time;
151             isHeartbeat = true;
152             if (pthread_create(&heartbeatThread, nullptr,
                    heartbeatFunction, this) != 0) {
153                 throw runtime_error("thread create error");
154             }
155         } else {
156             isHeartbeat = false;
157             if (pthread_join(heartbeatThread, nullptr) != 0) {
158                 throw runtime_error("thread join error");
159             }
160         }
161     }
162
163 private:
164     pid_t pid;
165     Tree t;
166     void *context;
167     Socket *publisher;
168     Socket *subscriber;
169     bool working;
170     pthread_t receiveMessage;
171 };
172
173
174 void *receiveFunction(void *server) {
175     auto *serverPointer = (Server *) server;
176     try {
177         pid_t child_pid = fork();
178         if (child_pid == -1) throw runtime_error("Can not fork.");
179         if (child_pid == 0) {
180             execl("client", "client", "0",
                    serverPointer->getPublisher()->getAddress().data(),
```

```
                nullptr);
181             throw runtime_error("Can not execl");
182         }
183         string address = createAddress(AddressType::PARENT_PUB,
                child_pid);
184         serverPointer->getSubscriber() = new
                Socket(serverPointer->getContext(),
                SocketType::SUBSCRIBER, address);
185         serverPointer->getTree().insert(0);
186         while(true) {
187             Message msg =
                    serverPointer->getSubscriber()->receive();
188             if (msg.command == CommandType::ERROR) {
189                 continue;
190             }
191             serverPointer->lastMessage = msg;
192             switch (msg.command) {
193                 case CommandType::CREATE_CHILD:
194                     cout << "OK: " << msg.getCreateIndex() <<
                            endl;
195                     break;
196                 case CommandType::RETURN:
197                     break;
198                 case CommandType::EXEC_CHILD:
199                     cout << "OK: response from node " <<
                            msg.getCreateIndex() << " is " <<
                            msg.value[0] << endl;
200                     break;
201                 default:
202                     break;
203             }
204         }
205     } catch (runtime_error &err) {
206         cout << "Server wasn't started " << err.what() << endl;
207     }
208     return nullptr;
209 }
210
211 void *heartbeatFunction(void *server) {
212     auto *serverPointer = (Server *) server;
213     while (serverPointer->isHeartbeat) {
214         vector<int> tmp = serverPointer->getTree().getElements();
215         bool answer = true;
216         for (int &e: tmp) {
217             if (!(serverPointer->check(e,
                    serverPointer->heartbeatTime))) {
218                 answer = false;
219                 cout << "Heartbeat: node " << e << " is
                        unavailable now\n";
220             }
```

```
221             }
222             if (answer) {
223                 cout << "OK\n";
224             }
225         }
226 }
227
228 Server *serverPointer = nullptr;
229
230 void terminate(int) {
231     if (serverPointer) {
232         serverPointer->~Server();
233     }
234     cout << to_string(getpid()) + " successfully terminated" <<
        endl;
235     exit(0);
236 }
237
238 int main() {
239     try {
240
241         // ctrl + C
242         if (signal(SIGINT, terminate) == SIG_ERR) {
243             throw runtime_error("Can not set SIGINT signal");
244         }
245
246         // segmentation fault
247         if (signal(SIGSEGV, terminate) == SIG_ERR) {
248             throw runtime_error("Can not set SIGSEGV signal");
249         }
250
251         // kill
252         if (signal(SIGTERM, terminate) == SIG_ERR) {
253             throw runtime_error("Can not set SIGTERM signal");
254         }
255
256         Server server = Server();
257         serverPointer = &server;
258         cout << getpid() << " server started correctly!\n";
259         while (true) {
260             try {
261                 string cmd;
262                 while (cin >> cmd) {
263                     server.commandProcessing(cmd);
264                 }
265             } catch (const runtime_error &arg) {
266                 cout << arg.what() << endl;
267             }
268         }
269     } catch (const runtime_error &arg) {
```

```
270             cout << arg.what() << endl;
271         } catch (...) {}
272         return 0;
273     }
```

Файл client.cpp:

```
1   #include <cstring>
2   #include <iostream>
3   #include <unistd.h>
4   #include <utility>
5   #include <vector>
6   #include <algorithm>
7   #include <csignal>
8   #include "headers/message.h"
9   #include "headers/socket.h"
10
11  using namespace std;
12
13  class Client {
14  private:
15      int id;
16      void *context;
17      bool terminated;
18
19  public:
20      Socket *childPublisherLeft;
21      Socket *childPublisherRight;
22      Socket *parentPublisher;
23      Socket *parentSubscriber;
24      Socket *leftSubscriber;
25      Socket *rightSubscriber;
26
27      Client(int id, const string& parentAddress) : id(id) {
28          context = createContext();
29          string address =
                createAddress(AddressType::CHILD_PUB_LEFT, getpid());
30          childPublisherLeft = new Socket(context,
                SocketType::PUBLISHER, address);
31          address = createAddress(AddressType::CHILD_PUB_RIGHT,
                getpid());
32          childPublisherRight = new Socket(context,
                SocketType::PUBLISHER, address);
33          address = createAddress(AddressType::PARENT_PUB,
                getpid());
34          parentPublisher = new Socket(context,
                SocketType::PUBLISHER, address);
35          parentSubscriber = new Socket(context,
                SocketType::SUBSCRIBER, parentAddress);
36          leftSubscriber = nullptr;
37          rightSubscriber = nullptr;
```

```
38          terminated = false;
39      }
40
41      ~Client() {
42          if (terminated) return;
43          terminated = true;
44          try {
45              delete childPublisherLeft;
46              delete childPublisherRight;
47              delete parentPublisher;
48              delete parentSubscriber;
49              delete leftSubscriber;
50              delete rightSubscriber;
51              destroyContext(context);
52          } catch (runtime_error &err) {
53              cout << "Server wasn't stopped " << err.what() <<
                    endl;
54          }
55      }
56
57      void messageProcessing(Message msg) {
58          switch (msg.command) {
59              case CommandType::ERROR:
60                  throw runtime_error("error message received");
61              case CommandType::RETURN: {
62                  msg.getToIndex() = SERVER_ID;
63                  sendUp(msg);
64                  break;
65              }
66              case CommandType::CREATE_CHILD: {
67                  msg.getCreateIndex() =
                        addChild(msg.getCreateIndex());
68                  msg.getToIndex() = SERVER_ID;
69                  sendUp(msg);
70                  break;
71              }
72              case CommandType::REMOVE_CHILD: {
73                  if (msg.withoutProcessing) {
74                      sendUp(msg);
75                      break;
76                  }
77                  if (msg.toIndex != getId() && msg.toIndex !=
                        UNIVERSAL_MESSAGE) {
78                      sendDown(msg);
79                      break;
80                  }
81                  msg.getToIndex() = UNIVERSAL_MESSAGE;
82                  sendDown(msg);
83                  this->~Client();
84                  throw invalid_argument("Exiting child...");
```

```
85              }
86              case CommandType::EXEC_CHILD: {
87                  double res = 0.0;
88                  for (int i = 0; i < msg.size; ++i) {
89                      res += msg.value[i];
90                  }
91                  msg.getToIndex() = SERVER_ID;
92                  msg.getCreateIndex() = getId();
93                  msg.value[0] = res;
94                  sendUp(msg);
95                  break;
96              }
97              default:
98                  throw runtime_error("undefined command");
99          }
100     }
101
102     void sendUp(Message msg) const {
103         msg.withoutProcessing = true;
104         parentPublisher->send(msg);
105     }
106
107     void sendDown(Message msg) const {
108         msg.withoutProcessing = false;
109         childPublisherLeft->send(msg);
110         childPublisherRight->send(msg);
111     }
112
113     int getId() const {
114         return id;
115     }
116
117     int addChild(int childId) {
118         pid_t pid = fork();
119         if (pid == -1) throw runtime_error("fork error");
120         if (!pid) {
121             string address;
122             if (childId < id) {
123                 address = childPublisherLeft->getAddress();
124             } else {
125                 address = childPublisherRight->getAddress();
126             }
127             execl("client", "client", to_string(childId).data(),
                    address.data(), nullptr);
128             throw runtime_error("execl error");
129         }
130         string address = createAddress(AddressType::PARENT_PUB,
                pid);
131         size_t timeout = 10000;
132         if (id > childId) {
```

```
133             leftSubscriber = new Socket(context,
                    SocketType::SUBSCRIBER, address);
134             zmq_setsockopt(leftSubscriber->getSocket(),
                    ZMQ_RCVTIMEO, &timeout, sizeof(timeout));
135         } else {
136             rightSubscriber = new Socket(context,
                    SocketType::SUBSCRIBER, address);
137             zmq_setsockopt(rightSubscriber->getSocket(),
                    ZMQ_RCVTIMEO, &timeout, sizeof(timeout));
138         }
139         return pid;
140     }
141
142 };
143
144 Client *clientPointer = nullptr;
145
146 void terminate(int) {
147     if (clientPointer) {
148         clientPointer->~Client();
149     }
150     cout << to_string(getpid()) + " successfully terminated" <<
            endl;
151     exit(0);
152 }
153
154 int main(int argc, char const *argv[]) {
155     if (argc != 3) {
156         cout << "-1" << endl;
157         return -1;
158     }
159     try {
160
161         // Ctrl + C
162         if (signal(SIGINT, terminate) == SIG_ERR) {
163             throw runtime_error("Can not set SIGINT signal");
164         }
165
166         // Segmentation fault
167         if (signal(SIGSEGV, terminate) == SIG_ERR) {
168             throw runtime_error("Can not set SIGSEGV signal");
169         }
170
171         // kill
172         if (signal(SIGTERM, terminate) == SIG_ERR) {
173             throw runtime_error("Can not set SIGTERM signal");
174         }
175
176         Client client(stoi(argv[1]), string(argv[2]));
177         clientPointer = &client;
```

```
178         cout << getpid() << ": client " << client.getId() << "
                successfully started" << endl;
179         while(true) {
180             Message msg = client.parentSubscriber->receive();
181             if (msg.toIndex != client.getId() && msg.toIndex !=
                    UNIVERSAL_MESSAGE) {
182                 if (msg.withoutProcessing) {
183                     client.sendUp(msg);
184                 } else {
185                     try {
186                         if (client.getId() < msg.toIndex) {
187                             msg.withoutProcessing = false;
188                             client.childPublisherRight->send(msg);
189                             msg =
                                    client.rightSubscriber->receive();
190                         } else {
191                             msg.withoutProcessing = false;
192                             client.childPublisherLeft->send(msg);
193                             msg =
                                    client.leftSubscriber->receive();
194                         }
195                         if (msg.command ==
                                CommandType::REMOVE_CHILD &&
                                msg.toIndex == PARENT_SIGNAL) {
196                             msg.toIndex = SERVER_ID;
197                             if (client.getId() <
                                    msg.getCreateIndex()) {
198                                 delete client.rightSubscriber;
199                                 client.rightSubscriber = nullptr;
200                             } else {
201                                 delete client.leftSubscriber;
202                                 client.leftSubscriber = nullptr;
203                             }
204                         }
205                         client.sendUp(msg);
206                     } catch (...) {
207                         client.sendUp(Message());
208                     }
209                 }
210             } else {
211                 clientPointer->messageProcessing(msg);
212             }
213         }
214     } catch (runtime_error &err) {
215         cout << getpid() << ": " << err.what() << '\n';
216     } catch (invalid_argument &inv) {
217         cout << getpid() << ": " << inv.what() << '\n';
218     }
219     return 0;
220 }
```

Файл message.h:

```
#ifndef _WRAP_ZMQ_H
#define _WRAP_ZMQ_H

#include <tuple>
#include <vector>
#include <atomic>
#include "zmq.h"

using namespace std;

#define UNIVERSAL_MESSAGE (-1)
#define SERVER_ID (-2)
#define PARENT_SIGNAL (-3)

enum struct SocketType {
    PUBLISHER,
    SUBSCRIBER,
};

enum struct CommandType {
    ERROR,
    RETURN,
    CREATE_CHILD,
    REMOVE_CHILD,
    EXEC_CHILD,
};

enum struct AddressType {
    CHILD_PUB_LEFT,
    CHILD_PUB_RIGHT,
    PARENT_PUB,
};

#define MAX_CAP 1000

class Message {
protected:
    static std::atomic<int> counter;
public:
    CommandType command = CommandType::ERROR;
    int toIndex;
    int createIndex;
    int uniqueIndex;
    bool withoutProcessing;
    int size = 0;
    double value[MAX_CAP] = {0};

    Message();

```

```
50      Message(CommandType command, int toIndex, int size, const
            double *value, int createIndex);
51
52      Message(CommandType new_command, int new_to_id, int new_id);
53
54      friend bool operator==(const Message &lhs, const Message
            &rhs);
55
56      int &getCreateIndex();
57
58      int &getToIndex();
59
60  };
61
62  void *createContext();
63
64  void destroyContext(void *context);
65
66  int getSocketType(SocketType type);
67
68  void *createSocket(void *context, SocketType type);
69
70  void closeSocket(void *socket);
71
72  string createAddress(AddressType type, pid_t id);
73
74  void bindSocket(void *socket, const string& address);
75
76  void unbindSocket(void *socket, const string& address);
77
78  void connectSocket(void *socket, const string& address);
79
80  void disconnectSocket(void *socket, const string& address);
81
82  void createMessage(zmq_msg_t *zmq_msg, Message &msg);
83
84  void sendMessage(void *socket, Message &msg);
85
86  Message getMessage(void *socket);
87
88  #endif
```

Файл message.cpp:

```
1  #include <tuple>
2  #include <cstring>
3  #include "headers/message.h"
4  #include <unistd.h>
5
6  using namespace std;
7
```

```
8  atomic<int> Message::counter;
9
10 Message::Message() {
11     command = CommandType::ERROR;
12     uniqueIndex = counter++;
13     withoutProcessing = false;
14 }
15
16 Message::Message(CommandType command, int toIndex, int size,
       const double *value, int createIndex)
17         : command(command), toIndex(toIndex), size(size),
               uniqueIndex(counter++), withoutProcessing(false),
18           createIndex(createIndex) {
19     for (int i = 0; i < size; ++i) {
20         this->value[i] = value[i];
21     }
22 }
23
24 Message::Message(CommandType command, int toIndex, int
       createIndex)
25         : command(command), toIndex(toIndex),
               uniqueIndex(counter++), withoutProcessing(false),
26           createIndex(createIndex) {}
27
28 bool operator==(const Message &lhs, const Message &rhs) {
29     return tie(lhs.command, lhs.toIndex, lhs.createIndex,
          lhs.uniqueIndex) ==
30             tie(rhs.command, rhs.toIndex, rhs.createIndex,
                   rhs.uniqueIndex);
31 }
32
33 int &Message::getCreateIndex() {
34     return createIndex;
35 }
36
37 int &Message::getToIndex() {
38     return toIndex;
39 }
40
41 void *createContext() {
42     void *context = zmq_ctx_new();
43     if (!context) {
44         throw runtime_error("unable to create new context");
45     }
46     return context;
47 }
48
49 void destroyContext(void *context) {
50     sleep(1);
51     if (zmq_ctx_destroy(context)) {
```

```
52          throw runtime_error("unable to destroy context");
53      }
54  }
55
56  int getSocketType(SocketType type) {
57      switch (type) {
58          case SocketType::PUBLISHER:
59              return ZMQ_PUB;
60          case SocketType::SUBSCRIBER:
61              return ZMQ_SUB;
62          default:
63              throw runtime_error("undefined socket type");
64      }
65  }
66
67  void *createSocket(void *context, SocketType type) {
68      int zmq_type = getSocketType(type);
69      void *socket = zmq_socket(context, zmq_type);
70      if (!socket) {
71          throw runtime_error("unable to create socket");
72      }
73      return socket;
74  }
75
76  void closeSocket(void *socket) {
77      sleep(1);
78      if (zmq_close(socket)) {
79          throw runtime_error("unable to close socket");
80      }
81  }
82
83  string createAddress(AddressType type, pid_t id) {
84      switch (type) {
85          case AddressType::PARENT_PUB:
86              return "ipc://parent_publisher_" + to_string(id);
87          case AddressType::CHILD_PUB_LEFT:
88              return "ipc://child_publisher_left_" + to_string(id);
89          case AddressType::CHILD_PUB_RIGHT:
90              return "ipc://child_publisher_right" + to_string(id);
91          default:
92              throw runtime_error("wrong address type");
93      }
94  }
95
96  void bindSocket(void *socket, const string& address) {
97      if (zmq_bind(socket, address.data())) {
98          throw runtime_error("unable to bind socket");
99      }
100 }
101
```

```
102 void unbindSocket(void *socket, const string& address) {
103     sleep(1);
104     if (zmq_unbind(socket, address.data())) {
105         throw runtime_error("unable to unbind socket");
106     }
107 }
108
109 void connectSocket(void *socket, const string& address) {
110     if (zmq_connect(socket, address.data())) {
111         throw runtime_error("unable to connect socket");
112     }
113     zmq_setsockopt(socket, ZMQ_SUBSCRIBE, nullptr, 0);
114 }
115
116 void disconnectSocket(void *socket, const string& address) {
117     if (zmq_disconnect(socket, address.data())) {
118         throw runtime_error("unable to disconnect socket.");
119     }
120 }
121
122 void createMessage(zmq_msg_t *zmq_msg, Message &msg) {
123     zmq_msg_init_size(zmq_msg, sizeof(msg));
124     memcpy(zmq_msg_data(zmq_msg), &msg, sizeof(msg));
125 }
126
127 void sendMessage(void *socket, Message &msg) {
128     zmq_msg_t zmq_msg;
129     createMessage(&zmq_msg, msg);
130     if (!zmq_msg_send(&zmq_msg, socket, 0)) {
131         throw runtime_error("unable to send message");
132     }
133     zmq_msg_close(&zmq_msg);
134 }
135
136 Message getMessage(void *socket) {
137     zmq_msg_t zmq_msg;
138     zmq_msg_init(&zmq_msg);
139     if (zmq_msg_recv(&zmq_msg, socket, 0) == -1) {
140         return {};
141     }
142     Message msg;
143     memcpy(&msg, zmq_msg_data(&zmq_msg), sizeof(msg));
144     zmq_msg_close(&zmq_msg);
145     return msg;
146 }
```

Файл socket.h:

```
1 #ifndef _SOCKET_H
2 #define _SOCKET_H
3
```

```
4  #include <string>
5  #include "message.h"
6
7  using namespace std;
8
9  class Socket {
10 public:
11     Socket(void *context, SocketType socketType, const string&
       address) :
12             socketType(socketType), address(address) {
13         socket = createSocket(context, socketType);
14         switch (socketType) {
15             case SocketType::PUBLISHER:
16                 bindSocket(socket, address);
17                 break;
18             case SocketType::SUBSCRIBER:
19                 connectSocket(socket, address);
20                 break;
21             default:
22                 throw logic_error("undefined connection type");
23         }
24     }
25
26     ~Socket() {
27         try {
28             switch (socketType) {
29                 case SocketType::PUBLISHER:
30                     unbindSocket(socket, address);
31                     break;
32                 case SocketType::SUBSCRIBER:
33                     disconnectSocket(socket, address);
34                     break;
35             }
36             closeSocket(socket);
37         } catch (exception& ex){
38             cout << "Socket wasn't closed: " << ex.what() << endl;
39         }
40     }
41
42     void send(Message message) {
43         if (socketType == SocketType::PUBLISHER){
44             sendMessage(socket, message);
45         } else {
46             throw logic_error("SUBSCRIBER can't send messages");
47         }
48     }
49
50     Message receive() {
51         if (socketType == SocketType::SUBSCRIBER){
52             return getMessage(socket);
```

```
53          } else {
54              throw logic_error("PUBLISHER can't receive messages");
55          }
56      }
57
58      string getAddress() const {
59          return address;
60      }
61
62      void *&getSocket() {
63          return socket;
64      }
65
66  private:
67      void *socket;
68      SocketType socketType;
69      string address;
70  };
71
72
73
74  #endif
```

2.2 Результат работы программы

```
1  matvey@matvey-Lenovo-IdeaPad-S340-15API:~/labs/2os/6lab/build$
       ./server
2  24719 server started correctly!
3  24723: client 0 successfully started
4  create 5
5  OK: 24726
6  24726: client 5 successfully started
7  create 6
8  OK: 24729
9  24729: client 6 successfully started
10 create 3
11 OK: 24732
12 24732: client 3 successfully started
13 exec 6 4 7 8 9 10
14 OK: response from node 6 is 34
15 exit
16 24729: Exiting child...
17 24732: Exiting child...
18 24723: Exiting child...
19 24726: Exiting child...
```

3 Вывод

В ходе выполнения лабораторной работы я познакомился с тем, как можно осуществлять межпроцессорное взаимодействие внутри приложения с помощью сокетов. Я узнал о том, что такое очередь сообщений и как её можно использовать, а также научился использовать библиотеку ZeroMQ для работы с вышеописанными сокетами. Для выполнения лабораторной работы я изучил основные паттерны организации взаимодействия между процессами на основе различных сокетов, а в приложении использовал топологию PUBLISHER — SUBSCRIBER.