Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Операционные системы»

Клиент для передачи мгновенных личных сообщений

Студент: О. В. Бабин
Преподаватель: А. А. Соколов
Группа: М8О-206Б-19
Дата: 30.12.2020
Оценка:
Подпись:

Москва, 2020

1 Постановка задачи

Цель проекта:

1. Приобретение практических навыков в использовании знаний, полученных в течении курса
2. Проведение исследования в выбранной предметной области

Задание:

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.

Выбранная тема: Создание клиента для передачи мгновенных личных сообщений.

Требуется создать клиент и сервер для обмена сообщениями между пользователями. Должна быть реализована регистрация и вход по логину/паролю, а также возможность отправки файлов весом до 1 Гб.

2 Общий метод и алгоритм решения

Проект будет состоять из сервера и клиета, собираться при помощи утилиты make. Работать будет на основе сервера сообщений ZMQ и сокетов ipc. Для работы с сервером сообщений буду использовать код, написанный в процессе работы над лабораторной 6-8. Клиент работает в двух потоках: первый - взаимодействие с пользователем и отправка сообщений, второй - получение сообщений. Сервер работает в одном потоке, реагируя на поступающие сообщения.

При запуске клиент дожидается ответа от сервера, затем устанавливается специально выделенная пара сокетов для взаимодействия сервера с конкретным клиентом. Каждому клиенту присваевается временный идентификатор. Затем пользователь либо регистрируется, либо входит по зарегистрированной паре логин-пароль. Затем пользователь выбирает, с кем хочет соединиться для переписки (по никнейму). В случае успеха появляется возможность обмена сообщениями и файлами.

Текстовые сообщения ограничены в длине 1000-ей знаков. Файлы отправляются пакетами по 1 Мб. Логин и пароль хранятся в зашифрованном виде.

3 Исходный код

m_zmq.h

```
1
2  #pragma once
3
4  #include <memory>
5  #include <string>
6
7  #include "message.h"
8
9  void* create_zmq_context();
10 void destroy_zmq_context(void* context);
11
12 enum class SocketType {
13   PUBLISHER,
14   SUBSCRIBER
15 };
16 void* create_zmq_socket(void* context, SocketType type);
17 void close_zmq_socket(void* socket);
18
19 const std::string ENDPOINT_PROTOCOL = "ipc://";
20 const std::string ENDPOINT_FOLDER = "tmp/";
21 enum class EndpointType {
22   SERVER_PUB_GENERAL,
23   SERVER_SUB_GENERAL,
24   SERVER_PUB,
25   CLIENT_PUB
26 };
27 std::string create_endpoint(EndpointType type, int id = 0);
28
29 void bind_zmq_socket(void* socket, std::string endpoint);
30 void unbind_zmq_socket(void* socket, std::string endpoint);
31 void connect_zmq_socket(void* socket, std::string endpoint);
32 void disconnect_zmq_socket(void* socket, std::string endpoint);
33
34 void send_zmq_msg(void* socket, std::shared_ptr<Message> msg);
35 std::shared_ptr<Message> get_zmq_msg(void* socket);
```

m_zmq.cpp

```
1
2  #include "m_zmq.h"
3
4  #include <unistd.h>
5  #include <zmq.h>
6
7  #include <algorithm>
8  #include <cstring>
```

```cpp
#include <iostream>

using namespace std;

void* create_zmq_context() {
  void* context = zmq_ctx_new();
  if (context == NULL) {
    throw runtime_error("Can't create new context. pid:" + to_string(getpid()));
  }
  return context;
}

void destroy_zmq_context(void* context) {
  if (zmq_ctx_destroy(context) != 0) {
    throw runtime_error("Can't destroy context. pid:" + to_string(getpid()));
  }
}

int get_zmq_socket_type(SocketType type) {
  switch (type) {
    case SocketType::PUBLISHER:
      return ZMQ_PUB;
    case SocketType::SUBSCRIBER:
      return ZMQ_SUB;
    default:
      throw logic_error("Undefined socket type");
  }
}

void* create_zmq_socket(void* context, SocketType type) {
  int zmq_type = get_zmq_socket_type(type);
  void* socket = zmq_socket(context, zmq_type);
  if (socket == NULL) {
    throw runtime_error("Can't create socket");
  }
  if (zmq_type == ZMQ_SUB) {
    if (zmq_setsockopt(socket, ZMQ_SUBSCRIBE, 0, 0) == -1) {
      throw runtime_error("Can't set ZMQ_SUBSCRIBE option");
    }
  }
  int linger_period = 0;
  if (zmq_setsockopt(socket, ZMQ_LINGER, &linger_period, sizeof(int)) == -1) {
    throw runtime_error("Can't set ZMQ_LINGER option");
  }
  return socket;
}

void close_zmq_socket(void* socket) {
  sleep(1); // Don't comment it, because sometimes zmq_close blocks
```

```
58    if (zmq_close(socket) != 0) {
59      throw runtime_error("Can't close socket");
60    }
61  }
62
63  string create_endpoint(EndpointType type, int id) {
64    switch (type) {
65      case EndpointType::SERVER_PUB_GENERAL:
66        return ENDPOINT_PROTOCOL + ENDPOINT_FOLDER + "server_pub_general"s;
67      case EndpointType::SERVER_SUB_GENERAL:
68        return ENDPOINT_PROTOCOL + ENDPOINT_FOLDER + "server_sub_general"s;
69      case EndpointType::SERVER_PUB:
70        return ENDPOINT_PROTOCOL + ENDPOINT_FOLDER + "server_pub_"s + to_string(id);
71      case EndpointType::CLIENT_PUB:
72        return ENDPOINT_PROTOCOL + ENDPOINT_FOLDER + "client_pub_"s + to_string(id);
73      default:
74        throw logic_error("Undefined endpoint type");
75    }
76  }
77
78  void bind_zmq_socket(void* socket, string endpoint) {
79    if (zmq_bind(socket, endpoint.data()) != 0) {
80      throw runtime_error("Can't bind socket (create 'tmp' folder)");
81    }
82  }
83
84  void unbind_zmq_socket(void* socket, string endpoint) {
85    if (zmq_unbind(socket, endpoint.data()) != 0) {
86      throw runtime_error("Can't unbind socket");
87    }
88  }
89
90  void connect_zmq_socket(void* socket, string endpoint) {
91    if (zmq_connect(socket, endpoint.data()) != 0) {
92      throw runtime_error("Can't connect socket (create 'tmp' folder)");
93    }
94  }
95
96  void disconnect_zmq_socket(void* socket, string endpoint) {
97    if (zmq_disconnect(socket, endpoint.data()) != 0) {
98      throw runtime_error("Can't disconnect socket");
99    }
100 }
101
102 void create_zmq_msg(zmq_msg_t* zmq_msg, shared_ptr<Message> msg_ptr) {
103   switch (msg_ptr->type()) {
104     case MessageType::BASIC:
105       zmq_msg_init_size(zmq_msg, sizeof(Message));
106       *(Message*)zmq_msg_data(zmq_msg) = *(Message*)msg_ptr.get();
```

```
107       break;
108     case MessageType::TEXT:
109       zmq_msg_init_size(zmq_msg, sizeof(TextMessage));
110       *(TextMessage*)zmq_msg_data(zmq_msg) = *(TextMessage*)msg_ptr.get();
111       break;
112     case MessageType::FILE:
113       zmq_msg_init_size(zmq_msg, sizeof(FileMessage));
114       *(FileMessage*)zmq_msg_data(zmq_msg) = *(FileMessage*)msg_ptr.get();
115       break;
116     default:
117       throw logic_error("Unemplemented message type");
118   }
119 }
120
121 void send_zmq_msg(void* socket, shared_ptr<Message> msg_ptr) {
122   zmq_msg_t zmq_msg;
123   create_zmq_msg(&zmq_msg, msg_ptr);
124   if (!zmq_msg_send(&zmq_msg, socket, 0)) {
125     throw runtime_error("Can't send message");
126   }
127   zmq_msg_close(&zmq_msg);
128 }
129
130 shared_ptr<Message> get_zmq_msg(void* socket) {
131   zmq_msg_t zmq_msg;
132   zmq_msg_init(&zmq_msg);
133   if (zmq_msg_recv(&zmq_msg, socket, 0) == -1) {
134     return Message::error_message();
135   }
136   shared_ptr<Message> msg_ptr = make_shared<Message>(*(Message*)zmq_msg_data(&zmq_msg)
        );
137
138   switch (msg_ptr->type()) {
139     case MessageType::BASIC:
140       break;
141     case MessageType::TEXT:
142       msg_ptr = make_shared<TextMessage>(*(TextMessage*)zmq_msg_data(&zmq_msg));
143       break;
144     case MessageType::FILE:
145       msg_ptr = make_shared<FileMessage>(*(FileMessage*)zmq_msg_data(&zmq_msg));
146       break;
147     default:
148       throw logic_error("Unemplemented message type");
149   }
150
151   zmq_msg_close(&zmq_msg);
152   return msg_ptr;
153 }
```

message.h

```cpp

#pragma once

#include <cstdint>
#include <memory>
#include <string>
#include <vector>

enum class MessageType {
  BASIC,
  TEXT,
  FILE
};

enum class CommandType {
  ERROR,
  CONNECT,
  DISCONNECT,
  TEXT,
  REGISTER,
  LOGIN,
  CREATE_CHAT,
  LEFT_CHAT,
  FILE_NAME,
  FILE_PART
};

class Message {
 protected:
  MessageType type_ = MessageType::BASIC;

 public:
  CommandType command = CommandType::ERROR;
  int from_id = 0;
  int to_id = 0;
  int value = 0;

  Message() = default;
  Message(CommandType command, int from_id, int to_id, int value);
  virtual ~Message() = default;

  std::string get_stats() const;
  MessageType type() const {
    return type_;
  }

  static std::shared_ptr<Message> error_message();
  static std::shared_ptr<Message> connect_message(int id);
  static std::shared_ptr<Message> disconnect_message(int id);
```

```
50    static std::shared_ptr<Message> left_chat_message(int id);
51  };
52
53  class TextMessage : public Message {
54   public:
55    static const size_t MAX_MESSAGE_SIZE = 1024;
56
57    char text[MAX_MESSAGE_SIZE + 1];
58
59    TextMessage();
60    TextMessage(CommandType command, int from_id, int to_id, const std::string& text,
          int value = 0);
61  };
62
63  class FileMessage : public Message {
64   public:
65    static const int COMMON_PART = 0;
66    static const int LAST_PART = 1;
67
68    static const size_t BUF_SIZE = 1000000;
69
70    uint8_t buf[BUF_SIZE];
71    size_t size;
72
73    FileMessage(CommandType command, int from_id, int to_id, int value, const std::
          vector<uint8_t>& buf_vec);
74  };
```

message.cpp

```
1
2   #include "message.h"
3
4   #include <unistd.h>
5
6   #include <cstring>
7   #include <stdexcept>
8
9   using namespace std;
10
11  Message::Message(CommandType command, int from_id, int to_id, int value)
12      : command(command),
13        from_id(from_id),
14        to_id(to_id),
15        value(value) {
16  }
17
18  std::string Message::get_stats() const {
19    return to_string(static_cast<int>(command)) + " " + to_string(from_id) + " " +
          to_string(to_id) + " " + to_string(value);
```

```
}
shared_ptr<Message> Message::error_message() {
  return make_shared<Message>(CommandType::ERROR, 0, 0, 0);
}
shared_ptr<Message> Message::connect_message(int id) {
  return make_shared<Message>(CommandType::CONNECT, id, 0, 0);
}
shared_ptr<Message> Message::disconnect_message(int id) {
  return make_shared<Message>(CommandType::DISCONNECT, id, 0, 0);
}
shared_ptr<Message> Message::left_chat_message(int id) {
  return make_shared<Message>(CommandType::LEFT_CHAT, id, 0, 0);
}

TextMessage::TextMessage() {
  type_ = MessageType::TEXT;
  text[0] = '\0';
}
TextMessage::TextMessage(CommandType command, int from_id, int to_id, const string&
    text_str, int value)
    : Message(command, from_id, to_id, value) {
  type_ = MessageType::TEXT;
  if (text_str.size() > MAX_MESSAGE_SIZE) {
    throw logic_error("Message text can't be longer, than MAX_MESSAGE_SIZE");
  }
  memcpy(text, text_str.data(), text_str.size() + 1);
}

FileMessage::FileMessage(CommandType command, int from_id, int to_id, int value, const
     vector<uint8_t>& buf_vec)
    : Message(command, from_id, to_id, value) {
  type_ = MessageType::FILE;
  if (buf_vec.size() > BUF_SIZE) {
    throw logic_error("File message size cannot be more than BUF_SIZE");
  }
  size = buf_vec.size();
  memcpy(buf, buf_vec.data(), buf_vec.size());
}
```

socket.h

```

#pragma once

#include <string>

#include "m_zmq.h"

enum class ConnectionType {
  BIND,
```

```
  CONNECT
};

class Socket {
 public:
  Socket(void* context, SocketType socket_type, std::string endpoint);
  ~Socket();

  void send(std::shared_ptr<Message> message);
  std::shared_ptr<Message> receive();

  void subscribe(std::string endpoint);
  void unsubscribe(std::string endpoint);
  std::string endpoint() const;

 private:
  void* socket_;
  SocketType socket_type_;
  std::string endpoint_;
};
```

client.h

```

#pragma once

#include <unistd.h>

#include <cstdint>
#include <filesystem>
#include <memory>
#include <string>
#include <vector>

#include "logger.h"
#include "socket.h"

class Client {
 public:
  Client();
  ~Client();

  void log(std::string message);
  void connect_to_server();
  void disconnect_from_server();

  void enter_in_system();
  void register_form();
  void login_form();

```

```
28    void send_text_msg(std::string message);
29    void send_file_msg(std::filesystem::path path);
30    void enter_chat(std::string uname);
31    void leave_chat();
32
33    int id() const;
34
35    enum class Status {
36      UNLOGGED,
37      LOGGED,
38      LOG_ERROR,
39      IN_CHAT
40    };
41    Status status = Status::UNLOGGED;
42
43    friend void* second_thread(void* cli_arg);
44
45   private:
46    int id_ = -1;
47    void* context_ = nullptr;
48    std::unique_ptr<Socket> publiser_;
49    std::unique_ptr<Socket> subscriber_;
50
51    pthread_t second_thread_id_;
52    bool server_is_avaible_ = false;
53
54    bool terminated_ = false;
55    Logger logger_ = Logger("log.txt");
56
57    void send(std::shared_ptr<Message> message);
58    std::shared_ptr<Message> receive();
59    void send_file_part_msg(const std::vector<uint8_t>& file_part, int value);
60
61    void leave_chat_actions();
62    std::ifstream fin;
63    std::ofstream fout;
64  };
```

client.cpp

```
1
2   #include "client.h"
3
4   #include <cstdlib>
5   #include <ctime>
6   #include <fstream>
7   #include <iostream>
8
9   #include "m_zmq.h"
10  #include "md5sum.h"
```

```
11
12 using namespace std;
13
14 const int ERR_LOOP = 2;
15 const int MESSAGE_WAITING_TIME = 1;
16
17 const string FILES_FOLDER = "files/";
18
19 void* second_thread(void* cli_arg) {
20   Client* client_ptr = (Client*)cli_arg;
21   try {
22     string endpoint = create_endpoint(EndpointType::SERVER_PUB_GENERAL);
23     client_ptr->subscriber_ = make_unique<Socket>(client_ptr->context_, SocketType::
         SUBSCRIBER, move(endpoint));
24
25     while (!client_ptr->terminated_) {
26       shared_ptr<Message> msg_ptr = client_ptr->receive();
27       if (msg_ptr->command == CommandType::ERROR) {
28         if (client_ptr->terminated_) {
29           return NULL;
30         } else {
31           cout << "Error" << endl;
32           continue;
33         }
34       }
35       if (msg_ptr->to_id != client_ptr->id()) {
36         continue;
37       }
38       client_ptr->log("Message received by client: "s + msg_ptr->get_stats());
39       switch (msg_ptr->command) {
40         case CommandType::CONNECT:
41           client_ptr->id_ = msg_ptr->value;
42           client_ptr->server_is_avaible_ = true;
43
44           endpoint = create_endpoint(EndpointType::CLIENT_PUB, client_ptr->id());
45           client_ptr->publiser_ = nullptr;
46           client_ptr->publiser_ = make_unique<Socket>(client_ptr->context_, SocketType
               ::PUBLISHER, move(endpoint));
47
48           endpoint = create_endpoint(EndpointType::SERVER_PUB, client_ptr->id());
49           client_ptr->subscriber_ = nullptr;
50           client_ptr->subscriber_ = make_unique<Socket>(client_ptr->context_,
               SocketType::SUBSCRIBER, move(endpoint));
51           break;
52         case CommandType::REGISTER:
53         case CommandType::LOGIN:
54           if (msg_ptr->value) {
55             client_ptr->status = Client::Status::LOGGED;
56           } else {
```

```
57            client_ptr->status = Client::Status::LOG_ERROR;
58          }
59          break;
60        case CommandType::CREATE_CHAT:
61          if (msg_ptr->value) {
62            cout << "You're in chat with: " << ((TextMessage*)msg_ptr.get())->text <<
                  endl;
63            client_ptr->status = Client::Status::IN_CHAT;
64          } else {
65            cout << "Can't create chat" << endl;
66          }
67          break;
68        case CommandType::LEFT_CHAT:
69          cout << "Companion left the chat" << endl;
70          cout << "Enter your companion username" << endl;
71          client_ptr->leave_chat_actions();
72          break;
73        case CommandType::TEXT:
74          cout << ">" << ((TextMessage*)msg_ptr.get())->text << endl;
75          break;
76        case CommandType::FILE_NAME: {
77          if (!filesystem::exists(FILES_FOLDER)) {
78            filesystem::create_directory(FILES_FOLDER);
79          }
80          string name = ((TextMessage*)msg_ptr.get())->text;
81          cout << "Filename: " << name << endl;
82          client_ptr->fout.open(FILES_FOLDER + name, ios::binary);
83          if (!client_ptr->fout.is_open()) {
84            throw runtime_error("Cannot open a file");
85          }
86          break;
87        }
88        case CommandType::FILE_PART: {
89          FileMessage& file_msg = *(FileMessage*)msg_ptr.get();
90          client_ptr->fout.write(reinterpret_cast<char*>(file_msg.buf), file_msg.size);
91          if (file_msg.value == FileMessage::LAST_PART) {
92            client_ptr->fout.close();
93            cout << "File is received" << endl;
94          }
95          break;
96        }
97        default:
98          throw logic_error("Undefined command type");
99          break;
100     }
101   }
102
103 } catch (exception& ex) {
104   client_ptr->log("Client exctption: "s + ex.what() + "\nTerminated by exception on
```

```
        client receive loop"s);
105     cout << "Terminated by error on loop" << endl;
106     exit(ERR_LOOP);
107   }
108   return NULL;
109 }
110
111 Client::Client() {
112   log("Starting client...");
113   context_ = create_zmq_context();
114
115   if (!filesystem::exists(ENDPOINT_FOLDER)) {
116     filesystem::create_directory(ENDPOINT_FOLDER);
117   }
118
119   string endpoint = create_endpoint(EndpointType::SERVER_SUB_GENERAL);
120   publiser_ = make_unique<Socket>(context_, SocketType::PUBLISHER, move(endpoint));
121
122   if (pthread_create(&second_thread_id_, 0, second_thread, this) != 0) {
123     cout << "Can't run second thread" << endl;
124     exit(ERR_LOOP);
125   }
126
127   srand(time(NULL) + clock());
128   id_ = rand();
129 }
130
131 Client::~Client() {
132   if (terminated_) {
133     log("Client double termination");
134     return;
135   }
136   disconnect_from_server();
137   log("Destroying client...");
138   terminated_ = true;
139
140   try {
141     publiser_ = nullptr;
142     subscriber_ = nullptr;
143     destroy_zmq_context(context_);
144     pthread_join(second_thread_id_, NULL);
145   } catch (exception& ex) {
146     log("Client wasn't destroyed: "s + ex.what());
147   }
148 }
149
150 int Client::id() const {
151   return id_;
152 }
```

```
153
154 void Client::send(shared_ptr<Message> message) {
155   publiser_->send(message);
156   log("Message sended from client: "s + message->get_stats());
157 }
158
159 shared_ptr<Message> Client::receive() {
160   return subscriber_->receive();
161 }
162
163 void Client::log(string message) {
164   logger_.log(move(message));
165 }
166
167 void Client::connect_to_server() {
168   while (!server_is_avaible_) {
169     cout << "Trying to connect to the server..." << endl;
170     send(Message::connect_message(id_));
171     sleep(MESSAGE_WAITING_TIME);
172   }
173   cout << "Connected to server" << endl;
174 }
175
176 void Client::disconnect_from_server() {
177   cout << "Disconnecting from the server..." << endl;
178   send(Message::disconnect_message(id_));
179 }
180
181 const int LOGIN_CHECK_COUNT = 5;
182 void Client::login_form() {
183   cout << "Enter login and password" << endl;
184   string uname, log, pas;
185   if (!(cin >> log >> pas)) {
186     throw runtime_error("Incorrect input");
187   }
188   status = Client::Status::UNLOGGED;
189   send(make_shared<TextMessage>(CommandType::LOGIN, id_, 0, md5sum(log) + " "s +
         md5sum(pas)));
190   int cnt = LOGIN_CHECK_COUNT;
191   while (cnt-- > 0 && status == Client::Status::UNLOGGED) {
192     cout << "Checking..." << endl;
193     sleep(1);
194   }
195   if (status == Client::Status::LOG_ERROR || status == Client::Status::UNLOGGED) {
196     cout << "Please, try again" << endl;
197   }
198 }
199
200 void Client::register_form() {
```

```
201   cout << "Enter username, login and password" << endl;
202   string uname, log, pas;
203   if (!(cin >> uname >> log >> pas)) {
204     throw runtime_error("Incorrect input");
205   }
206   status = Client::Status::UNLOGGED;
207   send(make_shared<TextMessage>(CommandType::REGISTER, id_, 0, uname + " "s + md5sum(
          log) + " "s + md5sum(pas)));
208   int cnt = LOGIN_CHECK_COUNT;
209   while (cnt-- > 0 && status == Client::Status::UNLOGGED) {
210     cout << "Checking..." << endl;
211     sleep(1);
212   }
213   if (status == Client::Status::LOG_ERROR || status == Client::Status::UNLOGGED) {
214     cout << "Please, try again" << endl;
215   }
216 }
217
218 void Client::enter_in_system() {
219   while (status != Client::Status::LOGGED) {
220     cout << "Do you have an account? (y/n)" << endl;
221     string str;
222     if (!(cin >> str)) {
223       throw runtime_error("Incorrect input");
224     }
225     if (str == "y") {
226       login_form();
227     } else if (str == "n") {
228       register_form();
229     } else {
230       cout << "Please, answer 'y' or 'n'" << endl;
231     }
232   }
233   cout << "Autorized" << endl;
234 }
235
236 void Client::send_text_msg(string message) {
237   send(make_shared<TextMessage>(CommandType::TEXT, id_, 0, move(message)));
238 }
239
240 void Client::send_file_part_msg(const vector<uint8_t>& file_part, int value) {
241   send(make_shared<FileMessage>(CommandType::FILE_PART, id_, 0, value, move(file_part)
          ));
242 }
243
244 void Client::send_file_msg(filesystem::path path) {
245   if (!filesystem::is_regular_file(path)) {
246     cout << "No such file" << endl;
247     return;
```

```
248   }
249   send(make_shared<TextMessage>(CommandType::FILE_NAME, id_, 0, path.filename()));
250   fin.open(path, ios::binary);
251   if (!fin.is_open()) {
252     cout << "Cannot open a file" << endl;
253     return;
254   }
255   vector<uint8_t> vec(FileMessage::BUF_SIZE);
256   size_t file_size = filesystem::file_size(path);
257   while (fin) {
258     fin.read(reinterpret_cast<char*>(vec.data()), FileMessage::BUF_SIZE);
259     vec.resize(fin.gcount());
260
261     if (fin) {
262       send_file_part_msg(vec, FileMessage::COMMON_PART);
263     } else {
264       send_file_part_msg(vec, FileMessage::LAST_PART);
265     }
266   }
267   fin.close();
268   cout << "File is sent" << endl;
269 }
270
271 void Client::enter_chat(string uname) {
272   cout << "Trying to create chat whith " << uname << "..." << endl;
273   send(make_shared<TextMessage>(CommandType::CREATE_CHAT, id_, 0, move(uname)));
274 }
275
276 void Client::leave_chat_actions() {
277   status = Client::Status::LOGGED;
278 }
279
280 void Client::leave_chat() {
281   cout << "Exiting from chat..." << endl;
282   send(Message::left_chat_message(id_));
283   leave_chat_actions();
284   cout << "Enter your companion username" << endl;
285 }
```

client_main.cpp

```
1
2  #include <signal.h>
3
4  #include <iostream>
5  #include <sstream>
6  #include <string>
7
8  #include "client.h"
9
```

```
10 using namespace std;
11
12 const int ERR_TERMINATED = 1;
13 const int UNIVERSAL_MESSAGE_ID = -256;
14
15 Client* client_ptr = nullptr;
16 void TerminateByUser(int) {
17   if (client_ptr != nullptr) {
18     client_ptr->~Client();
19     client_ptr->log("Client is terminated by user");
20   }
21   exit(0);
22 }
23
24 const string EXIT_COMMAND = "\\exit";
25 const string FILE_COMMAND = "\\file";
26
27 int main() {
28   try {
29     if (signal(SIGINT, TerminateByUser) == SIG_ERR) {
30       throw runtime_error("Can't set SIGINT signal");
31     }
32     if (signal(SIGSEGV, TerminateByUser) == SIG_ERR) {
33       throw runtime_error("Can't set SIGSEGV signal");
34     }
35
36     Client client;
37     client_ptr = &client;
38     client.log("Client is started correctly");
39
40     client.connect_to_server();
41     client.enter_in_system();
42
43     cout << "Enter your companion username" << endl;
44     string text;
45     while (getline(cin, text)) {
46       if (text == "") {
47         continue;
48       }
49       if (text.size() > TextMessage::MAX_MESSAGE_SIZE) {
50         cout << "Too long message" << endl;
51         continue;
52       }
53
54       if (client.status == Client::Status::IN_CHAT) {
55         if (text == EXIT_COMMAND) {
56           client.leave_chat();
57         } else if (text.size() > FILE_COMMAND.size() + 1 && text.substr(0, FILE_COMMAND
             .size()) == FILE_COMMAND && text[FILE_COMMAND.size()] == ' ') {
```

```
58          client.send_file_msg(text.substr(FILE_COMMAND.size() + 1));
59        } else {
60          client.send_text_msg(move(text));
61        }
62      } else {
63        client.enter_chat(move(text));
64      }
65    }
66
67  } catch (exception& ex) {
68    cout << (to_string(getpid()) + " Client exception: "s + ex.what() + "\nClient
          terminated by exception"s) << endl;
69    exit(ERR_TERMINATED);
70  }
71  return 0;
72 }
```

server.h

```
1
2  #pragma once
3
4  #include <unistd.h>
5
6  #include <memory>
7  #include <optional>
8  #include <unordered_map>
9
10 #include "logger.h"
11 #include "security.h"
12 #include "socket.h"
13
14 class Online {
15  public:
16   void add_user(std::string username, int id);
17   void remove_user(int id);
18   std::optional<int> get_id(std::string username) const;
19   std::optional<std::string> get_username(int id) const;
20   bool check_username(std::string username) const;
21
22  private:
23   std::unordered_map<int, std::string> id_to_username_;
24   std::unordered_map<std::string, int> username_to_id_;
25 };
26
27 class Rooms {
28  public:
29   bool add_room(int id1, int id2);
30   void remove_room(int id);
31   std::optional<int> get_companion(int id) const;
```

```
32    bool check_companion(int id) const;
33
34   private:
35    std::unordered_map<int, int> rooms_;
36  };
37
38  class Server {
39   public:
40    Server();
41    ~Server();
42
43    std::shared_ptr<Message> receive();
44
45    void log(std::string message);
46    void add_connection(int id);
47    void remove_connection(int id);
48    void register_form(std::shared_ptr<Message> msg_ptr);
49    void login_form(std::shared_ptr<Message> msg_ptr);
50    void create_room(int from_id, std::string username);
51    void exit_room(int id);
52    void send_from_user_to_user(int from_id, std::shared_ptr<Message> message);
53
54   private:
55    void* context_ = nullptr;
56    std::unique_ptr<Socket> subscriber_;
57    std::unique_ptr<Socket> general_publiser_;
58    std::unordered_map<int, std::unique_ptr<Socket>> id_to_publisher_;
59    Online online_;
60    Rooms rooms_;
61
62    int id_cntr = 0;
63
64    Logger logger_;
65    Security security;
66
67    void send_to_general(std::shared_ptr<Message> message);
68    void send_to_user(int id, std::shared_ptr<Message> message);
69  };
```

server.cpp

```
1
2   #include "server.h"
3
4   #include <pthread.h>
5   #include <signal.h>
6
7   #include <filesystem>
8   #include <iostream>
9   #include <sstream>
```

```
10
11 #include "m_zmq.h"
12
13 using namespace std;
14
15 void Online::add_user(std::string username, int id) {
16   id_to_username_[id] = username;
17   username_to_id_[move(username)] = id;
18 }
19 void Online::remove_user(int id) {
20   auto it = id_to_username_.find(id);
21   if (it != id_to_username_.end()) {
22     string user = it->second;
23     auto it2 = username_to_id_.find(move(user));
24     id_to_username_.erase(it);
25     username_to_id_.erase(it2);
26   } else {
27     cerr << "User to remove not found" << endl;
28   }
29 }
30 optional<int> Online::get_id(std::string username) const {
31   auto it = username_to_id_.find(move(username));
32   if (it != username_to_id_.end()) {
33     return it->second;
34   } else {
35     return nullopt;
36   }
37 }
38 optional<std::string> Online::get_username(int id) const {
39   auto it = id_to_username_.find(id);
40   if (it != id_to_username_.end()) {
41     return it->second;
42   } else {
43     return nullopt;
44   }
45 }
46 bool Online::check_username(std::string username) const {
47   auto it = username_to_id_.find(move(username));
48   return it != username_to_id_.end();
49 }
50
51 bool Rooms::add_room(int id1, int id2) {
52   if (rooms_.count(id1) > 0 || rooms_.count(id2) > 0) {
53     return false;
54   }
55   rooms_[id1] = id2;
56   rooms_[id2] = id1;
57   return true;
58 }
```

```
59 void Rooms::remove_room(int id) {
60   int id2 = rooms_[id];
61   rooms_.erase(id);
62   rooms_.erase(id2);
63 }
64 optional<int> Rooms::get_companion(int id) const {
65   auto it = rooms_.find(move(id));
66   if (it != rooms_.end()) {
67     return it->second;
68   } else {
69     return nullopt;
70   }
71 }
72 bool Rooms::check_companion(int id) const {
73   return (bool)get_companion(id);
74 }
75
76 Server::Server() {
77   log("Starting server...");
78   context_ = create_zmq_context();
79
80   if (!filesystem::exists(ENDPOINT_FOLDER)) {
81     filesystem::create_directory(ENDPOINT_FOLDER);
82   }
83
84   string endpoint = create_endpoint(EndpointType::SERVER_PUB_GENERAL);
85   general_publiser_ = make_unique<Socket>(context_, SocketType::PUBLISHER, move(
         endpoint));
86   endpoint = create_endpoint(EndpointType::SERVER_SUB_GENERAL);
87   subscriber_ = make_unique<Socket>(context_, SocketType::SUBSCRIBER, move(endpoint));
88 }
89
90 Server::~Server() {
91   log("Destroying server...");
92   try {
93     general_publiser_ = nullptr;
94     subscriber_ = nullptr;
95     for (auto& [_, ptr] : id_to_publisher_) {
96       ptr = nullptr;
97     }
98     destroy_zmq_context(context_);
99   } catch (exception& ex) {
100    log("Server wasn't destroyed: "s + ex.what());
101  }
102 }
103
104 void Server::send_to_general(shared_ptr<Message> message) {
105   general_publiser_->send(message);
106   log("Message sended from server: "s + message->get_stats());
```

```
107 }
108
109 void Server::send_to_user(int id, std::shared_ptr<Message> message) {
110   auto it = id_to_publisher_.find(id);
111   if (it == id_to_publisher_.end()) {
112     log("Message to id, that does not exist");
113     return;
114   }
115   message->from_id = 0;
116   message->to_id = id;
117   id_to_publisher_[id]->send(message);
118   log("Message sended from server to "s + to_string(id) + ": "s + message->get_stats()
        );
119 }
120
121 void Server::send_from_user_to_user(int from_id, std::shared_ptr<Message> message) {
122   auto opt = rooms_.get_companion(from_id);
123   if (!opt) {
124     send_to_user(from_id, Message::error_message());
125     return;
126   }
127   int to_id = *opt;
128   send_to_user(to_id, message);
129 }
130
131 shared_ptr<Message> Server::receive() {
132   shared_ptr<Message> message = subscriber_->receive();
133   log("Message received by server: "s + message->get_stats());
134   if (message->type() == MessageType::TEXT) {
135     log("Text: \""s + string(((TextMessage*)message.get())->text) + "\""s);
136   }
137   if (message->type() == MessageType::FILE) {
138     log("Package size: "s + to_string(((FileMessage*)message.get())->size));
139   }
140   return message;
141 }
142
143 void Server::log(std::string message) {
144   logger_.log(move(message));
145 }
146
147 void Server::add_connection(int id) {
148   int new_id = ++id_cntr;
149   string endpoint = create_endpoint(EndpointType::SERVER_PUB, new_id);
150   id_to_publisher_[new_id] = make_unique<Socket>(context_, SocketType::PUBLISHER, move
        (endpoint));
151   endpoint = create_endpoint(EndpointType::CLIENT_PUB, new_id);
152   subscriber_->subscribe(move(endpoint));
153
```

```
154   log("Connection added");
155   send_to_general(make_shared<Message>(CommandType::CONNECT, 0, id, new_id));
156 }
157
158 void print_bool(bool b) {
159   cout << b << endl;
160 }
161
162 void Server::create_room(int from_id, string username) {
163   auto opt = online_.get_id(username);
164   cout << *online_.get_username(from_id) << "a a" << username << "a" << endl;
165   if (!opt || rooms_.check_companion(from_id) || rooms_.check_companion(*opt) || *
          online_.get_username(from_id) == username) {
166     send_to_user(from_id, make_shared<Message>(CommandType::CREATE_CHAT, 0, from_id, 0)
          );
167     return;
168   }
169   int to_id = *opt;
170   rooms_.add_room(from_id, to_id);
171   send_to_user(from_id, make_shared<TextMessage>(CommandType::CREATE_CHAT, 0, from_id,
          *online_.get_username(to_id), 1));
172   send_to_user(to_id, make_shared<TextMessage>(CommandType::CREATE_CHAT, 0, to_id, *
          online_.get_username(from_id), 1));
173 }
174
175 void Server::exit_room(int id) {
176   auto opt = rooms_.get_companion(id);
177   if (opt) {
178     int companion_id = *opt;
179     send_to_user(companion_id, make_shared<Message>(CommandType::LEFT_CHAT, 0,
          companion_id, 0));
180   }
181   rooms_.remove_room(id);
182 }
183
184 void Server::remove_connection(int id) {
185   exit_room(id);
186   online_.remove_user(id);
187   id_to_publisher_.erase(id);
188   string endpoint = create_endpoint(EndpointType::CLIENT_PUB, id);
189   subscriber_->unsubscribe(move(endpoint));
190   log("Connection removed");
191 }
192
193 void Server::register_form(std::shared_ptr<Message> msg_ptr) {
194   TextMessage* text_msg_ptr = (TextMessage*)msg_ptr.get();
195   istringstream iss(text_msg_ptr->text);
196   string uname;
197   LogAndPas lap;
```

```
198   iss >> uname >> lap;
199   if (security.Register(uname, move(lap))) {
200     send_to_user(msg_ptr->from_id, make_shared<Message>(CommandType::REGISTER, 0,
          msg_ptr->from_id, 1));
201     online_.add_user(uname, msg_ptr->from_id);
202   } else {
203     send_to_user(msg_ptr->from_id, make_shared<Message>(CommandType::REGISTER, 0,
          msg_ptr->from_id, 0));
204   }
205 }
206
207 void Server::login_form(std::shared_ptr<Message> msg_ptr) {
208   TextMessage* text_msg_ptr = (TextMessage*)msg_ptr.get();
209   istringstream iss(text_msg_ptr->text);
210   LogAndPas lap;
211   iss >> lap;
212
213   auto opt_uname = security.get_username(move(lap));
214   if (opt_uname && !online_.check_username(*opt_uname)) {
215     send_to_user(msg_ptr->from_id, make_shared<Message>(CommandType::LOGIN, 0, msg_ptr
          ->from_id, 1));
216     online_.add_user(*opt_uname, msg_ptr->from_id);
217   } else {
218     send_to_user(msg_ptr->from_id, make_shared<Message>(CommandType::LOGIN, 0, msg_ptr
          ->from_id, 0));
219   }
220 }
```

server_main.cpp

```
1
2  #include <signal.h>
3
4  #include <filesystem>
5  #include <iostream>
6  #include <string>
7
8  #include "server.h"
9
10 using namespace std;
11
12 const int ERR_TERMINATED = 1;
13
14 Server* server_ptr = nullptr;
15 void TerminateByUser(int) {
16   if (server_ptr != nullptr) {
17     server_ptr->~Server();
18     server_ptr->log("Server is terminated by user");
19   }
20   exit(0);
```

```
}

void parse_cmd(Server& server, shared_ptr<Message> msg_ptr) {
  switch (msg_ptr->command) {
    case CommandType::CONNECT:
      server.add_connection(msg_ptr->from_id);
      break;
    case CommandType::DISCONNECT:
      server.remove_connection(msg_ptr->from_id);
      break;
    case CommandType::TEXT:
      if (msg_ptr->type() != MessageType::TEXT) {
        server.log("Text command in non text message");
        break;
      }
      server.send_from_user_to_user(msg_ptr->from_id, msg_ptr);
      break;
    case CommandType::REGISTER:
      if (msg_ptr->type() != MessageType::TEXT) {
        server.log("Register command in non text message");
        break;
      }
      server.register_form(msg_ptr);
      break;
    case CommandType::LOGIN:
      if (msg_ptr->type() != MessageType::TEXT) {
        server.log("Login command in non text message");
        break;
      }
      server.login_form(msg_ptr);
      break;
    case CommandType::CREATE_CHAT:
      if (msg_ptr->type() != MessageType::TEXT) {
        server.log("Register command in non text message");
        break;
      }
      server.create_room(msg_ptr->from_id, ((TextMessage*)msg_ptr.get())->text);
      break;
    case CommandType::LEFT_CHAT:
      server.exit_room(msg_ptr->from_id);
      break;
    case CommandType::FILE_NAME:
      if (msg_ptr->type() != MessageType::TEXT) {
        server.log("FileName command in non text message");
        break;
      }
      server.send_from_user_to_user(msg_ptr->from_id, msg_ptr);
      break;
    case CommandType::FILE_PART:
```

```
70        if (msg_ptr->type() != MessageType::FILE) {
71          server.log("FilePart command in non file message");
72          break;
73        }
74        server.send_from_user_to_user(msg_ptr->from_id, msg_ptr);
75        break;
76      default:
77        throw logic_error("Unimplemented command type");
78    }
79  }
80
81  int main() {
82    try {
83      if (signal(SIGINT, TerminateByUser) == SIG_ERR) {
84        throw runtime_error("Can't set SIGINT signal");
85      }
86      if (signal(SIGSEGV, TerminateByUser) == SIG_ERR) {
87        throw runtime_error("Can't set SIGSEGV signal");
88      }
89
90      Server server;
91      server_ptr = &server;
92      server.log("Server is started correctly");
93
94      while (true) {
95        shared_ptr<Message> msg_ptr = server.receive();
96        parse_cmd(server, msg_ptr);
97      }
98    } catch (exception& ex) {
99      cerr << ("Server exception: "s + ex.what() + "\nServer is terminated by exception")
           ;
100   }
101   return ERR_TERMINATED;
102 }
```

4 Пример работы

Сборка программы:

```
windicor@Lina-HP:~$ make
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c server_main.cpp
-o server_main.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c server.cpp -o
server.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c m_zmq.cpp -o
m_zmq.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c socket.cpp -o
socket.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c logger.cpp -o
logger.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c message.cpp
-o message.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c security.cpp
-o security.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c md5sum.cpp -o
md5sum.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable server_main.o server.o
m_zmq.o socket.o logger.o message.o security.o md5sum.o -o server -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c client_main.cpp
-o client_main.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c client.cpp -o
client.o -lzmq -lpthread
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable client_main.o client.o
m_zmq.o socket.o logger.o message.o security.o md5sum.o -o client -lzmq -lpthread
```

Запускаем сервер:

```
windicor@Lina-HP:~$ ./server
277: Starting server...
277: Server is started correctly
//тут происходит какая-то работа,в конце завершаем Ctrl+C
277: Destroying server...
277: Server is terminated by user
```

Запускаем клиент, входим, соединяемся с user2 (который также защёл в систему), обмениваемся парой сообщений, получаем файл и отправляем назад:

```
windicor@Lina-HP:~$ ./client
Trying to connect to the server...
Trying to connect to the server...
Connected to server
Do you have an account? (y/n)
y
Enter login and password
user qwerty123
Checking...
Autorized
Enter your companion username
user1
Trying to create chat whith user1...
You're in chat with: user1
Hi!
>Hi!
I'm user
>I'm user1
Filename: file_to_user
File is received
\file my_files/file_to_user
File is sent
\exit
Exiting from chat...
Enter your companion username
Disconnecting from the server...
```

5 Вывод

В процессе работы над курсовым проектом я повторил принципы работы с серверами сообщений (ZMQ), а также потоками и процессами. Реализация оказалась не так проста, как я изначально думал: трудности вызвало то, что сокеты на ZMQ однонаправленные, а также, в общем-то, не предназначены для отправки файлов. И если вторая проблема решилась довольно быстро, то решение первой заняло много времени. Было интересно писать достаточно больше приложение, так как в неём нужно учитывать много тонкостей, которые не важны при разработке небольших учебных программ. Также интересно было рассматривать возможные варианты поведения пользователя и системы. Также обратил внимание на тот факт, что если код пишется так, чтобы его можно было расширять и дополнять, то текст программы значительно растет.