Console Messenger

Generated by Doxygen 1.9.8

Глава 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Глава 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Глава 3

Class Documentation

3.1 ClientInfo Struct Reference

Информация о подключенном клиенте.

Public Attributes

- int fd
- std::string id
- std::string connected_to
- bool is_speaking = false
- std::string pending_request_from

3.1.1 Detailed Description

Информация о подключенном клиенте.

Definition at line 47 of file main_server.cpp.

3.1.2 Member Data Documentation

3.1.2.1 connected_to

ClientInfo::connected_to

ID клиента, с которым установлена беседа (пусто, если нет).

Definition at line 50 of file main_server.cpp.

3.1.2.2 fd

ClientInfo::fd

Дескриптор сокета клиента.

Definition at line 48 of file main_server.cpp.

3.1.2.3 id

ClientInfo::id

Идентификатор (Telegram ID) клиента.

Definition at line 49 of file main_server.cpp.

3.1.2.4 is_speaking

ClientInfo::is_speaking = false

Флаг права голоса (кто может отправлять сообщения).

Definition at line 51 of file main_server.cpp.

3.1.2.5 pending_request_from

ClientInfo::pending_request_from

Если не пусто — ID клиента, ожидающего подтверждения соединения.

Definition at line 52 of file main_server.cpp.

The documentation for this struct was generated from the following file:

- server/main_server.cpp

3.2 ServerConf Struct Reference

Параметры подключения к серверу.

Public Attributes

- std::string ip
- int port

3.2.1 Detailed Description

Параметры подключения к серверу.

Definition at line 45 of file main_client.cpp.

3.2.2 Member Data Documentation

3.2.2.1 ip

std::string ServerConf::ip

IPv4-адрес сервера.

Definition at line 46 of file main_client.cpp.

3.2.2.2 port

int ServerConf::port

Порт сервера.

Definition at line 47 of file main_client.cpp.

The documentation for this struct was generated from the following file:

- client/main_client.cpp

Глава 4

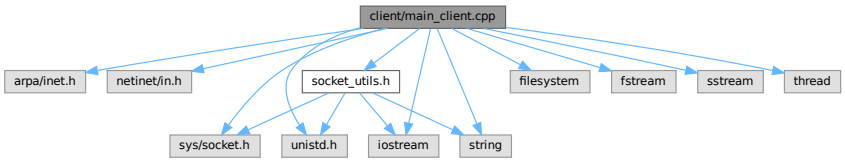
File Documentation

4.1 client/main_client.cpp File Reference

Клиент консольного мессенджера: подключение к серверу и обмен сообщениями.

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>
#include "socket_utils.h"
#include <filesystem>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <thread>
```

Include dependency graph for main_client.cpp:



Classes

- struct ServerConf

  Параметры подключения к серверу.

Functions

- bool valid_ip_port (const std::string &ip, int port)

  Проверить корректность IPv4-адреса и порта.
- ServerConf get_config ()

  Считать или запросить у пользователя настройки сервера.
- void receive_messages (int fd)

  Цикл приёма и вывода сообщений от сервера.
- int main ()

  Точка входа клиентского приложения.

Variables

- const std::string CFG_DIR = "CLIENT_SETTING"

  Директория для хранения конфигурационного файла.
- const std::string CFG_FILE = "CLIENT_SETTING/ip_port.txt"

  Путь к файлу с настройками (IP и порт сервера).

4.1.1 Detailed Description

Клиент консольного мессенджера: подключение к серверу и обмен сообщениями.

Программа читает конфигурацию сервера (IP и порт), устанавливает TCP-соединение, запускает поток для приёма сообщений и отправляет введённые пользователем строки.

Definition in file main_client.cpp.

4.1.2 Function Documentation

4.1.2.1 get_config()

ServerConf get_config ( )

Считать или запросить у пользователя настройки сервера.

Если файл с конфигурацией существует, пытается прочитать из него строку в формате "IP:порт". Если данные некорректны или файла нет, запрашивает ввод у пользователя до тех пор, пока не будет введена валидная пара. Сохраняет корректные настройки в файл.

Returns

Настройки сервера в виде ServerConf.

Definition at line 84 of file main_client.cpp.

```
00084                     {
00085     std::filesystem::create_directories(CFG_DIR);
00086     std::ifstream fin(CFG_FILE);
00087     std::string ip;
00088     int port;
00089     bool ok = false;
00090     if (fin) {
00091         std::getline(fin, ip, ':') && (fin >> port);
00092         ok = valid_ip_port(ip, port);
00093     }
00094     while (!ok) {
00095         std::cout << "Enter server IP: ";
00096         std::cin >> ip;
00097         std::cout << "Enter server port: ";
00098         std::cin >> port;
00099         std::cin.ignore();
00100         ok = valid_ip_port(ip, port);
00101         if (!ok)
00102             std::cout << "Invalid IP or port. Try again.\n";
00103     }
00104     std::ofstream(CFG_FILE, std::ios::trunc) << ip << ':' << port << '\n';
00105     return {ip, port};
00106 }
```

4.1.2.2 main()

int main ( )

Точка входа клиентского приложения.

Получает конфигурацию сервера, устанавливает TCP-соединение, запускает поток для приёма сообщений и в цикле отправляет введённые пользователем сообщения.

Returns

Код завершения (0 при успехе, иначе 1).

Definition at line 142 of file main_client.cpp.

```
00142         {
00143     ServerConf conf = get_config();
00144 
00145     int sock = socket(AF_INET, SOCK_STREAM, 0);
00146     if (sock == -1) {
00147         perror("socket");
00148         return 1;
00149     }
00150 
00151     sockaddr_in addr{};
00152     addr.sin_family = AF_INET;
00153     addr.sin_port = htons(conf.port);
00154     inet_pton(AF_INET, conf.ip.c_str(), &addr.sin_addr);
00155 
00156     if (connect(sock, (sockaddr*)&addr, sizeof(addr)) < 0) {
00157         perror("connect");
00158         return 1;
00159     }
00160 
00161     std::thread(receive_messages, sock).detach();
00162 
00163     std::string input;
00164     while (std::getline(std::cin, input)) {
00165         if (input.empty())
00166             continue;
00167         if (input.size() > MAX_LEN_INPUT) {
00168             std::cout « "Message longer than 2000 characters. Split it.\n";
00169             continue;
00170         }
00171         if (input == "/exit") {
00172             send_line(sock, "/exit");
00173             std::cout « "\nExiting...\n";
00174             break;
00175         }
00176         send_line(sock, input);
00177     }
00178     close(sock);
00179     return 0;
00180 }
```

4.1.2.3 receive_messages()

void receive_messages (
int fd)

Цикл приёма и вывода сообщений от сервера.

Читает строки из сокета через recv_line() до разрыва соединения. Выводит каждую строку на консоль. При получении специального маркера "*ENDM*" отображает приглашение ввода.

Parameters

fd	Дескриптор подключённого сокета сервера.

Definition at line 117 of file main_client.cpp.

```
00117                                {
00118     std::string line;
00119     while (recv_line(fd, line)) {
00120         if (line.empty())
00121             continue;
00122         if (line == "*ENDM*") {
00123             std::cout « "> " « std::flush;
00124             continue;
00125         }
00126         std::cout « line « '\n';
00127     }
00128     std::cout « "\nDisconnected from server.\n";
00129     close(fd);
00130     exit(0);
00131 }
```

4.1.2.4 valid_ip_port()

```
bool valid_ip_port (
                const std::string & ip,
                int port )
```

Проверить корректность IPv4-адреса и порта.

Использует inet_pton() для валидации формата IPv4 и проверяет, что порт находится в диапазоне 1..65535.

Parameters

ip	Строка с IPv4-адресом.
port	Номер порта.

Returns

true если адрес и порт валидны; false в противном случае.

See also

https://stackoverflow.com/questions/318236/how-do-you-validate-that-a-string-is-a-valid-ipv4-address-in-c

Note

Вдохновлено ответом ibodi, лицензия CC BY-SA 4.0.

Definition at line 67 of file main_client.cpp.

```
00067                                     {
00068     sockaddr_in tmp{};
00069     return inet_pton(AF_INET, ip.c_str(), &tmp.sin_addr) == 1 && port > 0 && port < 65536;
00070 }
```

4.1.3 Variable Documentation

4.1.3.1 CFG_DIR

```
const std::string CFG_DIR = "CLIENT_SETTING"
```

Директория для хранения конфигурационного файла.

Definition at line 31 of file main_client.cpp.

4.1.3.2 CFG_FILE

const std::string CFG_FILE = "CLIENT_SETTING/ip_port.txt"

Путь к файлу с настройками (IP и порт сервера).

Definition at line 36 of file main_client.cpp.

4.2 main_client.cpp

Go to the documentation of this file.

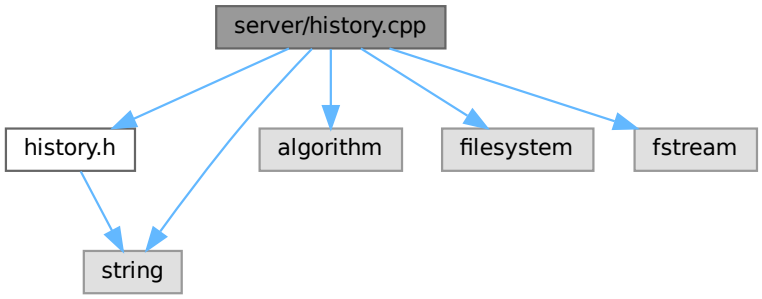
```
00001 /**
00002  * @file main_client.cpp
00003  * @brief Клиент консольного мессенджера: подключение к серверу и обмен сообщениями.
00004  *
00005  * Программа читает конфигурацию сервера (IP и порт),
00006  * устанавливает TCP-соединение, запускает поток
00007  * для приёма сообщений и отправляет введённые пользователем строки.
00008  */
00009
00010 #include <arpa/inet.h>
00011 #include <netinet/in.h>
00012 #include <sys/socket.h>
00013 #include <unistd.h>
00014
00015 #include "socket_utils.h"
00016 #include <filesystem>
00017 #include <fstream>
00018 #include <iostream>
00019 #include <sstream>
00020 #include <string>
00021 #include <thread>
00022
00023 /**
00024  * @brief Максимально допустимая длина сообщения от пользователя.
00025  */
00026 static const size_t MAX_LEN_INPUT = 2000;
00027
00028 /**
00029  * @brief Директория для хранения конфигурационного файла.
00030  */
00031 const std::string CFG_DIR = "CLIENT_SETTING";
00032
00033 /**
00034  * @brief Путь к файлу с настройками (IP и порт сервера).
00035  */
00036 const std::string CFG_FILE = "CLIENT_SETTING/ip_port.txt";
00037
00038 /**
00039  * @struct ServerConf
00040  * @brief Параметры подключения к серверу.
00041  *
00042  * @var ServerConf::ip   IPv4-адрес сервера.
00043  * @var ServerConf::port Порт сервера.
00044  */
00045 struct ServerConf {
00046     std::string ip; /**< IPv4-адрес сервера. */
00047     int port;       /**< Порт сервера. */
00048 };
00049
00050 /**
00051  * @brief Проверить корректность IPv4-адреса и порта.
00052  *
00053  * Использует inet_pton() для валидации формата IPv4
00054  * и проверяет, что порт находится в диапазоне 1..65535.
00055  *
00056  * @param ip    Строка с IPv4-адресом.
00057  * @param port  Номер порта.
00058  * @return true  если адрес и порт валидны;
00059  *         false в противном случае.
00060  *
00061  * @see https://stackoverflow.com/questions/318236/how-do-you-validate-that-a-string-is-a-valid-ipv4-address-in-c
00062  *
00063  * @note Вдохновлено ответом ibodi, лицензия CC BY-SA 4.0.
00064  */
00065
00066     // BEGIN: Borrowed code
00067 bool valid_ip_port(const std::string& ip, int port) {
```

```
00068     sockaddr_in tmp{};
00069     return inet_pton(AF_INET, ip.c_str(), &tmp.sin_addr) == 1 && port > 0 && port < 65536;
00070 }
00071     // END: Borrowed code
00072
00073 /**
00074  * @brief Считать или запросить у пользователя настройки сервера.
00075  *
00076  * Если файл с конфигурацией существует, пытается прочитать из него строку
00077  * в формате "IP:порт". Если данные некорректны или файла нет,
00078  * запрашивает ввод у пользователя до тех пор, пока не будет введена
00079  * валидная пара.
00080  * Сохраняет корректные настройки в файл.
00081  *
00082  * @return Настройки сервера в виде ServerConf.
00083  */
00084 ServerConf get_config() {
00085     std::filesystem::create_directories(CFG_DIR);
00086     std::ifstream fin(CFG_FILE);
00087     std::string ip;
00088     int port;
00089     bool ok = false;
00090     if (fin) {
00091         std::getline(fin, ip, ':') && (fin » port);
00092         ok = valid_ip_port(ip, port);
00093     }
00094     while (!ok) {
00095         std::cout « "Enter server IP: ";
00096         std::cin » ip;
00097         std::cout « "Enter server port: ";
00098         std::cin » port;
00099         std::cin.ignore();
00100         ok = valid_ip_port(ip, port);
00101         if (!ok)
00102             std::cout « "Invalid IP or port. Try again.\n";
00103     }
00104     std::ofstream(CFG_FILE, std::ios::trunc) « ip « ':' « port « '\n';
00105     return {ip, port};
00106 }
00107
00108 /**
00109  * @brief Цикл приёма и вывода сообщений от сервера.
00110  *
00111  * Читает строки из сокета через recv_line() до разрыва соединения.
00112  * Выводит каждую строку на консоль. При получении специального
00113  * маркера "*ENDM*" отображает приглашение ввода.
00114  *
00115  * @param fd Дескриптор подключённого сокета сервера.
00116  */
00117 void receive_messages(int fd) {
00118     std::string line;
00119     while (recv_line(fd, line)) {
00120         if (line.empty())
00121             continue;
00122         if (line == "*ENDM*") {
00123             std::cout « "> " « std::flush;
00124             continue;
00125         }
00126         std::cout « line « '\n';
00127     }
00128     std::cout « "\nDisconnected from server.\n";
00129     close(fd);
00130     exit(0);
00131 }
00132
00133 /**
00134  * @brief Точка входа клиентского приложения.
00135  *
00136  * Получает конфигурацию сервера, устанавливает TCP-соединение,
00137  * запускает поток для приёма сообщений и в цикле
00138  * отправляет введённые пользователем сообщения.
00139  *
00140  * @return Код завершения (0 при успехе, иначе 1).
00141  */
00142 int main() {
00143     ServerConf conf = get_config();
00144
00145     int sock = socket(AF_INET, SOCK_STREAM, 0);
00146     if (sock == -1) {
00147         perror("socket");
00148         return 1;
00149     }
00150
00151     sockaddr_in addr{};
00152     addr.sin_family = AF_INET;
00153     addr.sin_port = htons(conf.port);
00154     inet_pton(AF_INET, conf.ip.c_str(), &addr.sin_addr);
```

```
00155
00156     if (connect(sock, (sockaddr*)&addr, sizeof(addr)) < 0) {
00157         perror("connect");
00158         return 1;
00159     }
00160
00161     std::thread(receive_messages, sock).detach();
00162
00163     std::string input;
00164     while (std::getline(std::cin, input)) {
00165         if (input.empty())
00166             continue;
00167         if (input.size() > MAX_LEN_INPUT) {
00168             std::cout << "Message longer than 2000 characters. Split it.\n";
00169             continue;
00170         }
00171         if (input == "/exit") {
00172             send_line(sock, "/exit");
00173             std::cout << "\nExiting...\n";
00174             break;
00175         }
00176         send_line(sock, input);
00177     }
00178     close(sock);
00179     return 0;
00180 }
```

4.3 server/history.cpp File Reference

#include "history.h"
#include <algorithm>
#include <filesystem>
#include <fstream>
#include <string>
Include dependency graph for history.cpp:



Functions

- std::string get_history_filename (const std::string &user1, const std::string &user2)
- void ensure_history_folder_exists ()
- void append_message_to_history (const std::string &user1, const std::string &user2, const std↩ ::string &message)

  Добавить сообщение в историю чата двух пользователей.
- std::string load_history_for_users (const std::string &user1, const std::string &user2)

  Загрузить всю историю переписки между двумя пользователями.

4.3.1 Function Documentation

4.3.1.1 append_message_to_history()

```
void append_message_to_history (
                const std::string & user1,
                const std::string & user2,
                const std::string & message )
```

Добавить сообщение в историю чата двух пользователей.

Создаёт каталог HISTORY при необходимости и дописывает message в файл для пары пользователей.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.
message	Текст сообщения, включая символ новой строки.

Definition at line 22 of file history.cpp.

```
00023                                                   {
00024     ensure_history_folder_exists();
00025     std::ofstream file(get_history_filename(user1, user2), std::ios::app);
00026     if (file) {
00027         file << message;
00028     }
00029 }
```

4.3.1.2 ensure_history_folder_exists()

```
void ensure_history_folder_exists ( )
```

Definition at line 16 of file history.cpp.

```
00016                                   {
00017     if (!fs::exists("HISTORY")) {
00018         fs::create_directory("HISTORY");
00019     }
00020 }
```

4.3.1.3 get_history_filename()

```
std::string get_history_filename (
                const std::string & user1,
                const std::string & user2 )
```

Definition at line 9 of file history.cpp.

```
00009                                                                   {
00010     std::string u1 = user1, u2 = user2;
00011     if (u1 > u2)
00012         std::swap(u1, u2);
00013     return "HISTORY/history_" + u1 + "_" + u2 + ".txt";
00014 }
```

4.3.1.4 load_history_for_users()

```
std::string load_history_for_users (
                const std::string & user1,
                const std::string & user2 )
```

Загрузить всю историю переписки между двумя пользователями.

Открывает файл HISTORY/<min>___<max>.txt и возвращает его содержимое.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.

Returns

Строка с полным содержимым истории; пустая строка, если файл не существует или пуст.

Definition at line 31 of file history.cpp.

```
00031                                                                   {
00032     ensure_history_folder_exists();
00033     std::ifstream file(get_history_filename(user1, user2));
00034     std::string line, text;
00035     if (file.is_open()) {
00036         while (std::getline(file, line)){
00037             text += line + "\n";
00038         }
00039     }
00040     return text;
00041 }
```

4.4 history.cpp

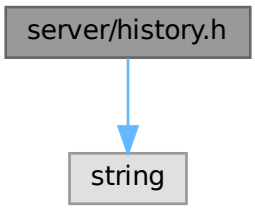
Go to the documentation of this file.

```
00001 #include "history.h"
00002 #include <algorithm>
00003 #include <filesystem>
00004 #include <fstream>
00005 #include <string>
00006
00007 namespace fs = std::filesystem;
00008
00009 std::string get_history_filename(const std::string& user1, const std::string& user2) {
00010     std::string u1 = user1, u2 = user2;
00011     if (u1 > u2)
00012         std::swap(u1, u2);
00013     return "HISTORY/history_" + u1 + "_" + u2 + ".txt";
00014 }
00015
00016 void ensure_history_folder_exists() {
00017     if (!fs::exists("HISTORY")) {
00018         fs::create_directory("HISTORY");
00019     }
00020 }
00021
00022 void append_message_to_history(const std::string& user1, const std::string& user2,
00023                                const std::string& message) {
00024     ensure_history_folder_exists();
00025     std::ofstream file(get_history_filename(user1, user2), std::ios::app);
00026     if (file) {
00027         file << message;
00028     }
00029 }
00030
00031 std::string load_history_for_users(const std::string& user1, const std::string& user2) {
00032     ensure_history_folder_exists();
00033     std::ifstream file(get_history_filename(user1, user2));
00034     std::string line, text;
00035     if (file.is_open()) {
00036         while (std::getline(file, line)){
00037             text += line + "\n";
00038         }
00039     }
00040     return text;
00041 }
```

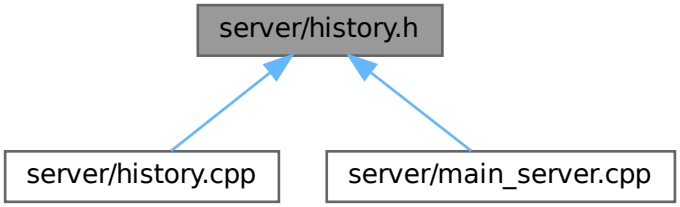
4.5 server/history.h File Reference

Работа с историей переписки между двумя пользователями.

#include <string>
Include dependency graph for history.h:



This graph shows which files directly or indirectly include this file:



Functions

- void append_message_to_history (const std::string &user1, const std::string &user2, const std::string &message)

  Добавить сообщение в историю чата двух пользователей.
- std::string load_history_for_users (const std::string &user1, const std::string &user2)

  Загрузить всю историю переписки между двумя пользователями.

4.5.1 Detailed Description

Работа с историей переписки между двумя пользователями.

Механизм:

- История хранится в каталоге HISTORY.
- Название файла истории для пары пользователей формируется лексикографически: HISTORY/<min>___<max>.txt.

Definition in file history.h.

4.5.2 Function Documentation

4.5.2.1 append_message_to_history()

```
void append_message_to_history (
                const std::string & user1,
                const std::string & user2,
                const std::string & message )
```

Добавить сообщение в историю чата двух пользователей.

Создаёт каталог HISTORY при необходимости и дописывает message в файл для пары пользователей.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.
message	Текст сообщения, включая символ новой строки.

Definition at line 22 of file history.cpp.

```
00023                                                 {
00024     ensure_history_folder_exists();
00025     std::ofstream file(get_history_filename(user1, user2), std::ios::app);
00026     if (file) {
00027         file « message;
00028     }
00029 }
```

4.5.2.2 load_history_for_users()

```
std::string load_history_for_users (
                const std::string & user1,
                const std::string & user2 )
```

Загрузить всю историю переписки между двумя пользователями.

Открывает файл HISTORY/<min>_ _ _<max>.txt и возвращает его содержимое.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.

Returns

Строка с полным содержимым истории; пустая строка, если файл не существует или пуст.

Definition at line 31 of file history.cpp.

```
00031                                                 {
00032     ensure_history_folder_exists();
00033     std::ifstream file(get_history_filename(user1, user2));
00034     std::string line, text;
00035     if (file.is_open()) {
00036         while (std::getline(file, line)){
```

```
00037             text += line + "\n";
00038         }
00039     }
00040     return text;
00041 }
```

4.6 history.h

Go to the documentation of this file.

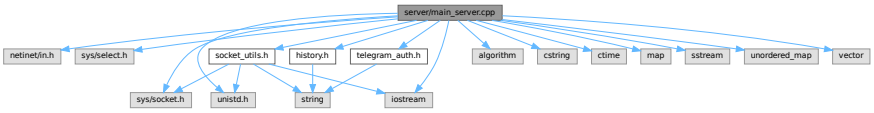
```
00001 /**
00002  * @file history.h
00003  * @brief Работа с историей переписки между двумя пользователями.
00004  *
00005  * Механизм:
00006  * - История хранится в каталоге HISTORY.
00007  * - Название файла истории для пары пользователей формируется
00008  *   лексикографически: HISTORY/<min>___<max>.txt.
00009  */
00010
00011 #ifndef HISTORY_H
00012 #define HISTORY_H
00013
00014 #include <string>
00015
00016 /**
00017  * @brief Добавить сообщение в историю чата двух пользователей.
00018  *
00019  * Создаёт каталог HISTORY при необходимости и дописывает @p message
00020  * в файл для пары пользователей.
00021  *
00022  * @param user1 Идентификатор первого пользователя.
00023  * @param user2 Идентификатор второго пользователя.
00024  * @param message Текст сообщения, включая символ новой строки.
00025  */
00026 void append_message_to_history(const std::string& user1, const std::string& user2,
00027                                const std::string& message);
00028
00029 /**
00030  * @brief Загрузить всю историю переписки между двумя пользователями.
00031  *
00032  * Открывает файл HISTORY/<min>___<max>.txt и возвращает его содержимое.
00033  *
00034  * @param user1 Идентификатор первого пользователя.
00035  * @param user2 Идентификатор второго пользователя.
00036  * @return Строка с полным содержимым истории; пустая строка,
00037  *         если файл не существует или пуст.
00038  */
00039 std::string load_history_for_users(const std::string& user1, const std::string& user2);
00040
00041 #endif  // HISTORY_H
```

4.7 server/main_server.cpp File Reference

Реализация сервера консольного мессенджера.

```
#include <netinet/in.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <unistd.h>
#include "telegram_auth.h"
#include "history.h"
#include "socket_utils.h"
#include <algorithm>
#include <cstring>
#include <ctime>
#include <iostream>
#include <map>
#include <sstream>
```

#include <unordered_map>
#include <vector>
Include dependency graph for main_server.cpp:



Classes

- struct ClientInfo

  Информация о подключенном клиенте.

Functions

- std::string get_timestamp ()

  Получить текущую дату и время.
- void disconnect_client (int fd, fd_set &master_fds)

  Отключить клиента и очистить его данные.
- void handle_client_command (int fd, const std::string &msg, fd_set &master_fds)

  Обработать команду клиента в режиме диалога.
- void handle_pending_response (int fd, const std::string &msg)

  Обработать ответ клиента на запрос соединения.
- int main ()

  Точка входа сервера.

Variables

- constexpr int PORT = 9090

  Порт, на котором слушает сервер.

4.7.1 Detailed Description

Реализация сервера консольного мессенджера.

Сервер принимает подключения клиентов по TCP, обеспечивает авторизацию через Telegram-коды, обработку команд клиентов (/connect, /vote, /end, /help, /exit, /shutdown), передачу сообщений между участниками и хранение истории.

Definition in file main_server.cpp.

4.7.2 Function Documentation

4.7.2.1 disconnect_client()

```
void disconnect_client (
            int fd,
            fd_set & master_fds )
```

Отключить клиента и очистить его данные.

Завершает соединение, удаляет из наборов клиентов, уведомляет партнера беседы.

Parameters

fd	Дескриптор сокета клиента для отключения.
master_fds	Ссылка на набор файловых дескрипторов select().

Definition at line 85 of file main_server.cpp.

```
00085                                           {
00086     if (clients.count(fd)) {
00087         std::string id = clients[fd].id;
00088         std::string connected_to = clients[fd].connected_to;
00089         std::cout « "\nDisconnecting client: " « id « " (fd: " « fd « ")\n";
00090 
00091         if (!connected_to.empty() && id_to_fd.count(connected_to)) {
00092             int target_fd = id_to_fd[connected_to];
00093             clients[target_fd].connected_to.clear();
00094             clients[target_fd].is_speaking = false;
00095             const std::string msg = "\nYour conversation partner has left the chat.\n";
00096             send_packet(target_fd, msg.c_str());
00097         }
00098 
00099         clients.erase(fd);
00100         id_to_fd.erase(id);
00101         FD_CLR(fd, &master_fds);
00102         close(fd);
00103     }
00104 }
```

4.7.2.2 get_timestamp()

std::string get_timestamp ( )

Получить текущую дату и время.

Возвращает строку в формате "YYYY-MM-DD HH:MM".

Returns

Форматированная метка времени.

Definition at line 69 of file main_server.cpp.

```
00069                                {
00070     time_t now = time(nullptr);
00071     char buf[20];
00072     strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M", localtime(&now));
00073     return std::string(buf);
00074 }
```

4.7.2.3 handle_client_command()

void handle_client_command (
int fd,
const std::string & msg,
fd_set & master_fds)

Обработать команду клиента в режиме диалога.

Поддерживаемые команды:

- /connect <ID>
- /vote
- /end
- /help
- /exit

Parameters

fd	Дескриптор сокета отправителя.
msg	Текст команды (без завершающего).
master_fds	Набор дескрипторов select() для обновления.

Definition at line 120 of file main_server.cpp.

```
00120                                                                 {
00121     if (msg.starts_with("/connect ")) {
00122         std::string target_id = msg.substr(9);
00123         if (id_to_fd.count(target_id)) {
00124             int target_fd = id_to_fd[target_id];
00125 
00126             if (!clients[target_fd].pending_request_from.empty()) {
00127                 send_packet(fd, "User is busy with another request.\n");
00128                 return;
00129             }
00130 
00131             if (!clients[target_fd].connected_to.empty()) {
00132                 const std::string notice = "\nUser '" + clients[fd].id +
00133                                            "' attempted to connect to you, but you are "
00134                                            "already in a conversation.\n";
00135                 send_packet(target_fd, notice.c_str());
00136                 send_packet(fd, "User is already connected.\n");
00137                 return;
00138             }
00139 
00140             clients[target_fd].pending_request_from = clients[fd].id;
00141             const std::string prompt = "\nUser '" + clients[fd].id + "' wants to connect. Accept? (yes/no)\n";
00142             send_packet(target_fd, prompt.c_str());
00143         } else {
00144             send_packet(fd, "User not found.\n");
00145         }
00146     } else if (msg == "/vote") {
00147         if (clients[fd].is_speaking) {
00148             std::string target_id = clients[fd].connected_to;
00149             if (!target_id.empty() && id_to_fd.count(target_id)) {
00150                 int target_fd = id_to_fd[target_id];
00151                 clients[fd].is_speaking = false;
00152                 clients[target_fd].is_speaking = true;
00153                 send_all(fd, "You passed the microphone.\n");
00154                 send_packet(target_fd, "You are now speaking.\n");
00155             } else {
00156                 send_packet(fd, "No connected client to pass speaking right.\n");
00157             }
00158         } else {
00159             send_packet(fd, "You are not the current speaker.\n");
00160         }
00161     } else if (msg == "/end") {
00162         std::string partner_id = clients[fd].connected_to;
00163         if (!partner_id.empty() && id_to_fd.count(partner_id)) {
00164             int partner_fd = id_to_fd[partner_id];
00165             clients[partner_fd].connected_to.clear();
00166             clients[partner_fd].is_speaking = false;
00167             send_packet(partner_fd, "\nYour conversation partner has ended the chat.\n");
00168         }
00169         clients[fd].connected_to.clear();
00170         clients[fd].is_speaking = false;
00171         send_packet(fd, "You have left the conversation.\n");
00172     } else if (msg == "/help") {
00173         const std::string help =
00174             "Available commands:\n"
00175             "/connect <ID> - request chat with user\n"
00176             "/vote         - pass speaker role\n"
00177             "/end          - end current conversation\n"
00178             "/exit         - exit the chat completely\n"
00179             "/help         - show this message\n";
00180         send_packet(fd, help.c_str());
00181     } else if (msg == "/exit") {
00182         disconnect_client(fd, master_fds);
00183     } else {
00184         send_packet(fd, "Only /connect <ID>, /vote, /end, /exit, /help are allowed.\n");
00185     }
00186 }
```

4.7.2.4 handle_pending_response()

void handle_pending_response (

int fd,
const std::string & msg)

Обработать ответ клиента на запрос соединения.

Если клиент ранее отправил /connect и ожидает ответа, эта функция устанавливает связь и пересылает историю.

Parameters

fd	Дескриптор сокета отвечающего клиента.
msg	Сообщение-ответ ("yes"/"no").

Definition at line 197 of file main_server.cpp.

```
00197                                                 {
00198     ClientInfo& responder = clients[fd];
00199     if (responder.pending_request_from.empty())
00200         return;
00201 
00202     std::string requester_id = responder.pending_request_from;
00203     responder.pending_request_from.clear();
00204 
00205     if (!id_to_fd.count(requester_id)) {
00206         send_packet(fd, "Requester disconnected.\n");
00207         return;
00208     }
00209 
00210     int requester_fd = id_to_fd[requester_id];
00211     if (msg == "yes") {
00212         std::cout « "Clients connected: " « responder.id « " <-> " « requester_id « std::endl;
00213         responder.connected_to = requester_id;
00214         clients[requester_fd].connected_to = responder.id;
00215         clients[requester_fd].is_speaking = true;
00216 
00217         std::string history = load_history_for_users(responder.id, requester_id);
00218         if (!history.empty()) {
00219             send_all(fd, "Chat history:\n");
00220             send_all(fd, history.c_str());
00221             send_all(requester_fd, "Chat history:\n");
00222             send_all(requester_fd, history.c_str());
00223         }
00224         send_packet(requester_fd, "Connection accepted. You are now speaking.\n");
00225         send_all(fd, "Connection established. You are a listener.\n");
00226     } else {
00227         send_packet(requester_fd, "Connection rejected.\n");
00228         send_packet(fd, "Connection declined.\n");
00229     }
00230 }
```

4.7.2.5 main()

int main ( )

Точка входа сервера.

Запускает прослушивание порта, обрабатывает подключения и команды до получения /shutdown.

Returns

0 при корректном завершении, иначе код ошибки.

Definition at line 240 of file main_server.cpp.

```
00240           {
00241     ensure_bot_token();
00242 
00243     int listener = socket(AF_INET, SOCK_STREAM, 0);
00244     if (listener == -1) {
```

```
00245         perror("socket");
00246         return 1;
00247     }
00248
00249     int opt = 1;
00250     setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
00251
00252     sockaddr_in server_addr{};
00253     server_addr.sin_family = AF_INET;
00254     server_addr.sin_port = htons(PORT);
00255     server_addr.sin_addr.s_addr = INADDR_ANY;
00256
00257     if (bind(listener, (sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
00258         perror("bind");
00259         return 1;
00260     }
00261
00262     listen(listener, SOMAXCONN);
00263
00264     std::cout « "Server listening on port " « PORT « std::endl;
00265
00266     fd_set master_fds, read_fds;
00267     FD_ZERO(&master_fds);
00268     FD_SET(listener, &master_fds);
00269     FD_SET(STDIN_FILENO, &master_fds);
00270     int fd_max = listener;
00271
00272     while (true) {
00273         read_fds = master_fds;
00274         if (select(fd_max + 1, &read_fds, nullptr, nullptr, nullptr) == -1) {
00275             perror("select");
00276             break;
00277         }
00278
00279
00280         for (int fd = 0; fd <= fd_max; ++fd) {
00281             if (!FD_ISSET(fd, &read_fds))
00282                 continue;
00283
00284             if (fd == STDIN_FILENO) {
00285                 std::string cmd;
00286                 std::getline(std::cin, cmd);
00287                 if (cmd == "/shutdown") {
00288                     std::cout « "Shutting down server...\n";
00289                         // BEGIN: Borrowed code
00290                     for (auto& [cfd, info] : clients)
00291                         send_all(cfd, "\nServer is shutting down.\n");
00292                     for (auto& [cfd, info] : clients)
00293                         close(cfd);
00294                         // END: Borrowed code
00295                     close(listener);
00296                     std::cout « "Server stopped.\n";
00297                     return 0;
00298                 }
00299                 continue;
00300             }
00301
00302             if (fd == listener) {
00303                 int client_fd = accept(listener, nullptr, nullptr);
00304                 if (client_fd != -1) {
00305                     std::cout « "New client connected, fd: " « client_fd « std::endl;
00306                     FD_SET(client_fd, &master_fds);
00307                     fd_max = std::max(fd_max, client_fd);
00308                     const char* ask_id = "Enter your ID\n";
00309                     send_packet(client_fd, ask_id);
00310                 }
00311             } else {
00312                 std::string msg;
00313                 if (!recv_line(fd, msg)) {
00314                     disconnect_client(fd, master_fds);
00315                     continue;
00316                 }
00317
00318                 if (clients.count(fd) == 0 && !pending_auth.count(fd)) {
00319                     std::string chat_id = msg;
00320                     if (chat_id.empty()) {
00321                         send_packet(fd, "Chat ID cannot be empty. Try again\n");
00322                         continue;
00323                     }
00324
00325                     std::string code = generate_auth_code();
00326                     if (send_telegram_code(chat_id, code)) {
00327                         pending_auth[fd] = chat_id;
00328                         const char* sent = "Telegram code sent. Enter the code to log in\n";
00329                         send_packet(fd, sent);
00330                     } else {
00331                         send_packet(fd,
```

```
00332                             "Failed to send Telegram message.\nUse command /exit to "
00333                             "exit.\nCheck the telegram ID and write it again");
00334                 }
00335             }
00336
00337             else if (pending_auth.count(fd)) {
00338                 std::string entered_code = msg;
00339                 std::string chat_id = pending_auth[fd];
00340                 if (verify_auth_code(chat_id, entered_code)) {
00341                     if (id_to_fd.count(chat_id)) {
00342                         int old_fd = id_to_fd[chat_id];
00343                         send_packet(old_fd, "\nYou have been logged out (second login detected).\n");
00344                         disconnect_client(old_fd, master_fds);
00345                     }
00346
00347                     clients[fd] = ClientInfo{fd, chat_id};
00348                     std::cout « "Client authorized: " « chat_id « " (fd: " « fd « ")" « std::endl;
00349                     id_to_fd[chat_id] = fd;
00350                     pending_auth.erase(fd);
00351
00352                     std::string welcome =
00353                         "Welcome, " + chat_id + "! Use /connect <ID>, /vote, /end, /exit, /help\n";
00354                     send_packet(fd, welcome.c_str());
00355                 } else {
00356                     send_packet(fd, "Incorrect code. Try again\n");
00357                 }
00358             }
00359
00360             else if (!clients[fd].pending_request_from.empty()) {
00361                 handle_pending_response(fd, msg);
00362             }
00363
00364             else if (!msg.empty() && msg[0] == '/') {
00365                 handle_client_command(fd, msg, master_fds);
00366             }
00367
00368             else {
00369                 if (clients[fd].connected_to.empty()) {
00370                     send_packet(fd,
00371                                 "You are not in a conversation.\nUse /connect <ID> to "
00372                                 "start chatting.\n");
00373                     continue;
00374                 }
00375                 if (!clients[fd].is_speaking) {
00376                     send_all(fd,
00377                             "You cannot send messages unless you're the current "
00378                             "speaker.\n");
00379                     continue;
00380                 }
00381
00382                 std::string target_id = clients[fd].connected_to;
00383                 if (!target_id.empty() && id_to_fd.count(target_id)) {
00384                     int target_fd = id_to_fd[target_id];
00385                     std::string timestamp = get_timestamp();
00386                     std::string sender = clients[fd].id;
00387                     std::string text = "[" + timestamp + "] " + sender + ": " + msg + "\n";
00388                     send_all(target_fd, text.c_str());
00389                     append_message_to_history(sender, target_id, text);
00390                 } else {
00391                     send_packet(fd, "Not connected. Use /connect <ID>\n");
00392                 }
00393             }
00394         }
00395     }
00396 }
00397
00398 close(listener);
00399 return 0;
00400 }
```

4.7.3 Variable Documentation

4.7.3.1 PORT

constexpr int PORT = 9090 [constexpr]

Порт, на котором слушает сервер.

Definition at line 30 of file main_server.cpp.

4.8 main_server.cpp

Go to the documentation of this file.

```
00001 /**
00002  * @file main_server.cpp
00003  * @brief Реализация сервера консольного мессенджера.
00004  *
00005  * Сервер принимает подключения клиентов по TCP, обеспечивает
00006  * авторизацию через Telegram-коды, обработку команд клиентов
00007  * (/connect, /vote, /end, /help, /exit, /shutdown),
00008  * передачу сообщений между участниками и хранение истории.
00009  */
00010
00011 #include <netinet/in.h>
00012 #include <sys/select.h>
00013 #include <sys/socket.h>
00014 #include <unistd.h>
00015
00016 #include "telegram_auth.h"
00017
00018 #include "history.h"
00019 #include "socket_utils.h"
00020 #include <algorithm>
00021 #include <cstring>
00022 #include <ctime>
00023 #include <iostream>
00024 #include <map>
00025 #include <sstream>
00026 #include <unordered_map>
00027 #include <vector>
00028
00029 /// Порт, на котором слушает сервер.
00030 constexpr int PORT = 9090;
00031
00032 /**
00033  * @struct ClientInfo
00034  * @brief Информация о подключенном клиенте.
00035  *
00036  * @var ClientInfo::fd
00037  * Дескриптор сокета клиента.
00038  * @var ClientInfo::id
00039  * Идентификатор (Telegram ID) клиента.
00040  * @var ClientInfo::connected_to
00041  * ID клиента, с которым установлена беседа (пусто, если нет).
00042  * @var ClientInfo::is_speaking
00043  * Флаг права голоса (кто может отправлять сообщения).
00044  * @var ClientInfo::pending_request_from
00045  * Если не пусто — ID клиента, ожидающего подтверждения соединения.
00046  */
00047 struct ClientInfo {
00048     int fd;
00049     std::string id;
00050     std::string connected_to;
00051     bool is_speaking = false;
00052     std::string pending_request_from;
00053 };
00054
00055 /// Карта: дескриптор сокета -> информация о клиенте.
00056 static std::unordered_map<int, ClientInfo> clients;
00057 /// Карта: Telegram ID клиента -> дескриптор сокета.
00058 static std::unordered_map<std::string, int> id_to_fd;
00059 /// Карта: дескриптор сокета -> Telegram ID (ожидающие код).
00060 static std::unordered_map<int, std::string> pending_auth;
00061
00062 /**
00063  * @brief Получить текущую дату и время.
00064  *
00065  * Возвращает строку в формате "YYYY-MM-DD HH:MM".
00066  *
00067  * @return Форматированная метка времени.
00068  */
00069 std::string get_timestamp() {
00070     time_t now = time(nullptr);
00071     char buf[20];
00072     strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M", localtime(&now));
00073     return std::string(buf);
00074 }
00075
00076 /**
00077  * @brief Отключить клиента и очистить его данные.
00078  *
00079  * Завершает соединение, удаляет из наборов клиентов,
00080  * уведомляет партнера беседы.
00081  *
00082  * @param fd Дескриптор сокета клиента для отключения.
```

```
00083  * @param master_fds Ссылка на набор файловых дескрипторов select().
00084  */
00085 void disconnect_client(int fd, fd_set& master_fds) {
00086     if (clients.count(fd)) {
00087         std::string id = clients[fd].id;
00088         std::string connected_to = clients[fd].connected_to;
00089         std::cout « "\nDisconnecting client: " « id « " (fd: " « fd « ")\n";
00090 
00091         if (!connected_to.empty() && id_to_fd.count(connected_to)) {
00092             int target_fd = id_to_fd[connected_to];
00093             clients[target_fd].connected_to.clear();
00094             clients[target_fd].is_speaking = false;
00095             const std::string msg = "\nYour conversation partner has left the chat.\n";
00096             send_packet(target_fd, msg.c_str());
00097         }
00098 
00099         clients.erase(fd);
00100         id_to_fd.erase(id);
00101         FD_CLR(fd, &master_fds);
00102         close(fd);
00103     }
00104 }
00105 
00106 /**
00107  * @brief Обработать команду клиента в режиме диалога.
00108  *
00109  * Поддерживаемые команды:
00110  *  - /connect <ID>
00111  *  - /vote
00112  *  - /end
00113  *  - /help
00114  *  - /exit
00115  *
00116  * @param fd   Дескриптор сокета отправителя.
00117  * @param msg  Текст команды (без завершающего \n).
00118  * @param master_fds Набор дескрипторов select() для обновления.
00119  */
00120 void handle_client_command(int fd, const std::string& msg, fd_set& master_fds) {
00121     if (msg.starts_with("/connect ")) {
00122         std::string target_id = msg.substr(9);
00123         if (id_to_fd.count(target_id)) {
00124             int target_fd = id_to_fd[target_id];
00125 
00126             if (!clients[target_fd].pending_request_from.empty()) {
00127                 send_packet(fd, "User is busy with another request.\n");
00128                 return;
00129             }
00130 
00131             if (!clients[target_fd].connected_to.empty()) {
00132                 const std::string notice = "\nUser '" + clients[fd].id +
00133                                            "' attempted to connect to you, but you are "
00134                                            "already in a conversation.\n";
00135                 send_packet(target_fd, notice.c_str());
00136                 send_packet(fd, "User is already connected.\n");
00137                 return;
00138             }
00139 
00140             clients[target_fd].pending_request_from = clients[fd].id;
00141             const std::string prompt = "\nUser '" + clients[fd].id + "' wants to connect. Accept? (yes/no)\n";
00142             send_packet(target_fd, prompt.c_str());
00143         } else {
00144             send_packet(fd, "User not found.\n");
00145         }
00146     } else if (msg == "/vote") {
00147         if (clients[fd].is_speaking) {
00148             std::string target_id = clients[fd].connected_to;
00149             if (!target_id.empty() && id_to_fd.count(target_id)) {
00150                 int target_fd = id_to_fd[target_id];
00151                 clients[fd].is_speaking = false;
00152                 clients[target_fd].is_speaking = true;
00153                 send_all(fd, "You passed the microphone.\n");
00154                 send_packet(target_fd, "You are now speaking.\n");
00155             } else {
00156                 send_packet(fd, "No connected client to pass speaking right.\n");
00157             }
00158         } else {
00159             send_packet(fd, "You are not the current speaker.\n");
00160         }
00161     } else if (msg == "/end") {
00162         std::string partner_id = clients[fd].connected_to;
00163         if (!partner_id.empty() && id_to_fd.count(partner_id)) {
00164             int partner_fd = id_to_fd[partner_id];
00165             clients[partner_fd].connected_to.clear();
00166             clients[partner_fd].is_speaking = false;
00167             send_packet(partner_fd, "\nYour conversation partner has ended the chat.\n");
00168         }
00169         clients[fd].connected_to.clear();
```

```
00170         clients[fd].is_speaking = false;
00171         send_packet(fd, "You have left the conversation.\n");
00172     } else if (msg == "/help") {
00173         const std::string help =
00174             "Available commands:\n"
00175             "/connect <ID> - request chat with user\n"
00176             "/vote         - pass speaker role\n"
00177             "/end          - end current conversation\n"
00178             "/exit         - exit the chat completely\n"
00179             "/help         - show this message\n";
00180         send_packet(fd, help.c_str());
00181     } else if (msg == "/exit") {
00182         disconnect_client(fd, master_fds);
00183     } else {
00184         send_packet(fd, "Only /connect <ID>, /vote, /end, /exit, /help are allowed.\n");
00185     }
00186 }
00187
00188 /**
00189  * @brief Обработать ответ клиента на запрос соединения.
00190  *
00191  * Если клиент ранее отправил /connect и ожидает ответа,
00192  * эта функция устанавливает связь и пересылает историю.
00193  *
00194  * @param fd  Дескриптор сокета отвечающего клиента.
00195  * @param msg Сообщение-ответ ("yes"/"no").
00196  */
00197 void handle_pending_response(int fd, const std::string& msg) {
00198     ClientInfo& responder = clients[fd];
00199     if (responder.pending_request_from.empty())
00200         return;
00201
00202     std::string requester_id = responder.pending_request_from;
00203     responder.pending_request_from.clear();
00204
00205     if (!id_to_fd.count(requester_id)) {
00206         send_packet(fd, "Requester disconnected.\n");
00207         return;
00208     }
00209
00210     int requester_fd = id_to_fd[requester_id];
00211     if (msg == "yes") {
00212         std::cout « "Clients connected: " « responder.id « " <-> " « requester_id « std::endl;
00213         responder.connected_to = requester_id;
00214         clients[requester_fd].connected_to = responder.id;
00215         clients[requester_fd].is_speaking = true;
00216
00217         std::string history = load_history_for_users(responder.id, requester_id);
00218         if (!history.empty()) {
00219             send_all(fd, "Chat history:\n");
00220             send_all(fd, history.c_str());
00221             send_all(requester_fd, "Chat history:\n");
00222             send_all(requester_fd, history.c_str());
00223         }
00224         send_packet(requester_fd, "Connection accepted. You are now speaking.\n");
00225         send_all(fd, "Connection established. You are a listener.\n");
00226     } else {
00227         send_packet(requester_fd, "Connection rejected.\n");
00228         send_packet(fd, "Connection declined.\n");
00229     }
00230 }
00231
00232 /**
00233  * @brief Точка входа сервера.
00234  *
00235  * Запускает прослушивание порта,
00236  * обрабатывает подключения и команды до получения /shutdown.
00237  *
00238  * @return 0 при корректном завершении, иначе код ошибки.
00239  */
00240 int main() {
00241     ensure_bot_token();
00242
00243     int listener = socket(AF_INET, SOCK_STREAM, 0);
00244     if (listener == -1) {
00245         perror("socket");
00246         return 1;
00247     }
00248
00249     int opt = 1;
00250     setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
00251
00252     sockaddr_in server_addr{};
00253     server_addr.sin_family = AF_INET;
00254     server_addr.sin_port = htons(PORT);
00255     server_addr.sin_addr.s_addr = INADDR_ANY;
00256
```

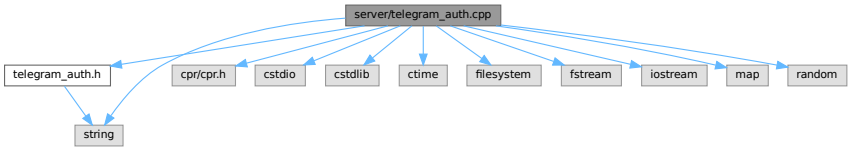
```
00257     if (bind(listener, (sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
00258         perror("bind");
00259         return 1;
00260     }
00261 
00262     listen(listener, SOMAXCONN);
00263 
00264     std::cout « "Server listening on port " « PORT « std::endl;
00265 
00266     fd_set master_fds, read_fds;
00267     FD_ZERO(&master_fds);
00268     FD_SET(listener, &master_fds);
00269     FD_SET(STDIN_FILENO, &master_fds);
00270     int fd_max = listener;
00271 
00272     while (true) {
00273         read_fds = master_fds;
00274         if (select(fd_max + 1, &read_fds, nullptr, nullptr, nullptr) == -1) {
00275             perror("select");
00276             break;
00277         }
00278 
00279 
00280         for (int fd = 0; fd <= fd_max; ++fd) {
00281             if (!FD_ISSET(fd, &read_fds))
00282                 continue;
00283 
00284             if (fd == STDIN_FILENO) {
00285                 std::string cmd;
00286                 std::getline(std::cin, cmd);
00287                 if (cmd == "/shutdown") {
00288                     std::cout « "Shutting down server...\n";
00289                         // BEGIN: Borrowed code
00290                     for (auto& [cfd, info] : clients)
00291                         send_all(cfd, "\nServer is shutting down.\n");
00292                     for (auto& [cfd, info] : clients)
00293                         close(cfd);
00294                         // END: Borrowed code
00295                     close(listener);
00296                     std::cout « "Server stopped.\n";
00297                     return 0;
00298                 }
00299                 continue;
00300             }
00301 
00302             if (fd == listener) {
00303                 int client_fd = accept(listener, nullptr, nullptr);
00304                 if (client_fd != -1) {
00305                     std::cout « "New client connected, fd: " « client_fd « std::endl;
00306                     FD_SET(client_fd, &master_fds);
00307                     fd_max = std::max(fd_max, client_fd);
00308                     const char* ask_id = "Enter your ID\n";
00309                     send_packet(client_fd, ask_id);
00310                 }
00311             } else {
00312                 std::string msg;
00313                 if (!recv_line(fd, msg)) {
00314                     disconnect_client(fd, master_fds);
00315                     continue;
00316                 }
00317 
00318                 if (clients.count(fd) == 0 && !pending_auth.count(fd)) {
00319                     std::string chat_id = msg;
00320                     if (chat_id.empty()) {
00321                         send_packet(fd, "Chat ID cannot be empty. Try again\n");
00322                         continue;
00323                     }
00324 
00325                     std::string code = generate_auth_code();
00326                     if (send_telegram_code(chat_id, code)) {
00327                         pending_auth[fd] = chat_id;
00328                         const char* sent = "Telegram code sent. Enter the code to log in\n";
00329                         send_packet(fd, sent);
00330                     } else {
00331                         send_packet(fd,
00332                                     "Failed to send Telegram message.\nUse command /exit to "
00333                                     "exit.\nCheck the telegram ID and write it again");
00334                     }
00335                 }
00336 
00337                 else if (pending_auth.count(fd)) {
00338                     std::string entered_code = msg;
00339                     std::string chat_id = pending_auth[fd];
00340                     if (verify_auth_code(chat_id, entered_code)) {
00341                         if (id_to_fd.count(chat_id)) {
00342                             int old_fd = id_to_fd[chat_id];
00343                             send_packet(old_fd, "\nYou have been logged out (second login detected).\n");
```

```
00344                     disconnect_client(old_fd, master_fds);
00345                 }
00346
00347                 clients[fd] = ClientInfo{fd, chat_id};
00348                 std::cout « "Client authorized: " « chat_id « " (fd: " « fd « ")" « std::endl;
00349                 id_to_fd[chat_id] = fd;
00350                 pending_auth.erase(fd);
00351
00352                 std::string welcome =
00353                     "Welcome, " + chat_id + "! Use /connect <ID>, /vote, /end, /exit, /help\n";
00354                 send_packet(fd, welcome.c_str());
00355             } else {
00356                 send_packet(fd, "Incorrect code. Try again\n");
00357             }
00358         }
00359
00360         else if (!clients[fd].pending_request_from.empty()) {
00361             handle_pending_response(fd, msg);
00362         }
00363
00364         else if (!msg.empty() && msg[0] == '/') {
00365             handle_client_command(fd, msg, master_fds);
00366         }
00367
00368         else {
00369             if (clients[fd].connected_to.empty()) {
00370                 send_packet(fd,
00371                             "You are not in a conversation.\nUse /connect <ID> to "
00372                             "start chatting.\n");
00373                 continue;
00374             }
00375             if (!clients[fd].is_speaking) {
00376                 send_all(fd,
00377                          "You cannot send messages unless you're the current "
00378                          "speaker.\n");
00379                 continue;
00380             }
00381
00382             std::string target_id = clients[fd].connected_to;
00383             if (!target_id.empty() && id_to_fd.count(target_id)) {
00384                 int target_fd = id_to_fd[target_id];
00385                 std::string timestamp = get_timestamp();
00386                 std::string sender = clients[fd].id;
00387                 std::string text = "[" + timestamp + "] " + sender + ": " + msg + "\n";
00388                 send_all(target_fd, text.c_str());
00389                 append_message_to_history(sender, target_id, text);
00390             } else {
00391                 send_packet(fd, "Not connected. Use /connect <ID>\n");
00392             }
00393         }
00394     }
00395 }
00396 }
00397
00398 close(listener);
00399 return 0;
00400 }
```

4.9 server/telegram_auth.cpp File Reference

```
#include "telegram_auth.h"
#include <cpr/cpr.h>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <map>
#include <random>
#include <string>
```

Include dependency graph for telegram_auth.cpp:



Functions

- std::string generate_auth_code ()

  Сгенерировать случайный шестизначный код для авторизации.
- void ensure_bot_token ()

  Убедиться, что глобальный токен бота загружен.
- bool send_telegram_code (const std::string &chat_id, const std::string &code)

  Отправить код авторизации через Telegram Bot API.
- bool verify_auth_code (const std::string &chat_id, const std::string &code)

  Проверить введённый пользователем код авторизации.

Variables

- std::string BOT_TOKEN
- std::map< std::string, std::string > auth_codes

4.9.1 Function Documentation

4.9.1.1 ensure_bot_token()

void ensure_bot_token ( )

Убедиться, что глобальный токен бота загружен.

При первом вызове пытается считать токен из файла SERVER_SETTINGS/BOT_TOKEN.txt. Если файл отсутствует или пуст, создаёт его и выводит сообщение об ошибке. Завершает программу при отсутствии токена.

Definition at line 30 of file telegram_auth.cpp.

```
00030                          {
00031     namespace fs = std::filesystem;
00032     fs::path dir = "SERVER_SETTINGS";
00033     fs::path file = dir / "BOT_TOKEN.txt";
00034 
00035     if (!fs::exists(dir))
00036         fs::create_directories(dir);
00037 
00038     if (fs::exists(file)) {
00039         std::ifstream in(file);
00040         std::getline(in, BOT_TOKEN);
00041     } else {
00042         std::ofstream out(file);
00043     }
00044 
00045     if (BOT_TOKEN.empty()) {
00046         std::cerr « "[TelegramAuth] Файл " « file
00047                   « " не содержит токен.\n"
00048                     "Добавьте токен в первую строку и перезапустите сервер.\n";
00049         exit(1);
00050     }
00051 }
```

4.9.1.2 generate_auth_code()

```
std::string generate_auth_code ( )
```

Сгенерировать случайный шестизначный код для авторизации.

Код состоит из цифр [0-9] и всегда имеет длину 6 символов.

Returns

Сгенерированный код (например, "042517").

Definition at line 17 of file telegram_auth.cpp.

```
00017                                {
00018     static bool seeded = false;
00019     if (!seeded) {
00020         std::srand(static_cast<unsigned int>(std::time(nullptr)));
00021         seeded = true;
00022     }
00023
00024     std::string digits = "0123456789", code;
00025     for (int i = 0; i < 6; ++i)
00026         code += digits[std::rand() % 10];
00027     return code;
00028 }
```

4.9.1.3 send_telegram_code()

```
bool send_telegram_code (
                const std::string & chat_id,
                const std::string & code )
```

Отправить код авторизации через Telegram Bot API.

Формирует и выполняет HTTP-запрос для отправки сообщения с одноразовым кодом.

Parameters

chat_id	Идентификатор Telegram-чата получателя.
code	Шестизначный код, который будет отправлен.

Returns

true, если сообщение успешно отправлено и код сохранён; false в случае ошибки при вызове curl или неверного ответа API.

Definition at line 53 of file telegram_auth.cpp.

```
00053                                                    {
00054     cpr::Response response = cpr::Post(
00055         cpr::Url{"https://api.telegram.org/bot" + BOT_TOKEN + "/sendMessage"},
00056         cpr::Payload{{"chat_id", chat_id}, {"text", "Your code is: " + code}}
00057     );
00058     if (response.text.find("\"ok\":true") != std::string::npos) {
00059         auth_codes[chat_id] = code;
00060         return true;
00061     }
00062     return false;
00063 }
```

4.9.1.4 verify_auth_code()

```
bool verify_auth_code (
                const std::string & chat_id,
                const std::string & code )
```

Проверить введённый пользователем код авторизации.

Сравнивает переданный code с сохранённым в auth_codes для данного chat_id.

Parameters

chat↩ _id	Идентификатор Telegram-чата, для которого код генерировался.
code	Код, введённый пользователем.

Returns

true, если код совпадает с ранее сгенерированным; иначе false.

Definition at line 65 of file telegram_auth.cpp.

```
00065                                                   {
00066     return auth_codes.count(chat_id) && auth_codes[chat_id] == code;
00067 }
```

4.9.2 Variable Documentation

4.9.2.1 auth_codes

```
std::map<std::string, std::string> auth_codes
```

Definition at line 15 of file telegram_auth.cpp.

4.9.2.2 BOT_TOKEN

```
std::string BOT_TOKEN
```

Definition at line 13 of file telegram_auth.cpp.

4.10 telegram_auth.cpp

Go to the documentation of this file.

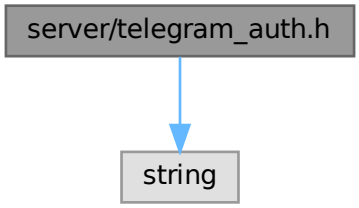
```
00001 #include "telegram_auth.h"
00002 #include <cpr/cpr.h>
00003 #include <cstdio>
00004 #include <cstdlib>
00005 #include <ctime>
00006 #include <filesystem>
00007 #include <fstream>
00008 #include <iostream>
00009 #include <map>
00010 #include <random>
00011 #include <string>
00012
```

```
00013 std::string BOT_TOKEN;
00014
00015 std::map<std::string, std::string> auth_codes;
00016
00017 std::string generate_auth_code() {
00018     static bool seeded = false;
00019     if (!seeded) {
00020         std::srand(static_cast<unsigned int>(std::time(nullptr)));
00021         seeded = true;
00022     }
00023
00024     std::string digits = "0123456789", code;
00025     for (int i = 0; i < 6; ++i)
00026         code += digits[std::rand() % 10];
00027     return code;
00028 }
00029
00030 void ensure_bot_token() {
00031     namespace fs = std::filesystem;
00032     fs::path dir = "SERVER_SETTINGS";
00033     fs::path file = dir / "BOT_TOKEN.txt";
00034
00035     if (!fs::exists(dir))
00036         fs::create_directories(dir);
00037
00038     if (fs::exists(file)) {
00039         std::ifstream in(file);
00040         std::getline(in, BOT_TOKEN);
00041     } else {
00042         std::ofstream out(file);
00043     }
00044
00045     if (BOT_TOKEN.empty()) {
00046         std::cerr « "[TelegramAuth] Файл " « file
00047                 « " не содержит токен.\n"
00048                   "Добавьте токен в первую строку и перезапустите сервер.\n";
00049         exit(1);
00050     }
00051 }
00052
00053 bool send_telegram_code(const std::string& chat_id, const std::string& code) {
00054     cpr::Response response = cpr::Post(
00055         cpr::Url{"https://api.telegram.org/bot" + BOT_TOKEN + "/sendMessage"},
00056         cpr::Payload{{"chat_id", chat_id}, {"text", "Your code is: " + code}}
00057     );
00058     if (response.text.find("\"ok\":true") != std::string::npos) {
00059         auth_codes[chat_id] = code;
00060         return true;
00061     }
00062     return false;
00063 }
00064
00065 bool verify_auth_code(const std::string& chat_id, const std::string& code) {
00066     return auth_codes.count(chat_id) && auth_codes[chat_id] == code;
00067 }
```

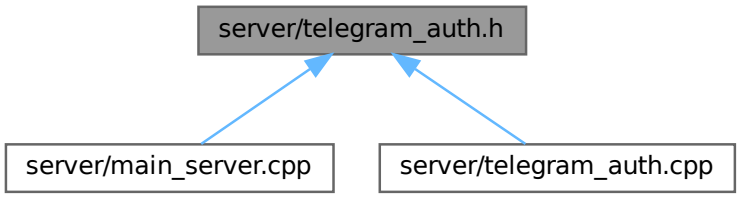
4.11 server/telegram_auth.h File Reference

Интерфейс для Telegram-аутентификации: генерация, отправка и проверка кодов.

#include <string>
Include dependency graph for telegram_auth.h:



This graph shows which files directly or indirectly include this file:



Functions

- std::string generate_auth_code ()
  Сгенерировать случайный шестизначный код для авторизации.
- bool send_telegram_code (const std::string &chat_id, const std::string &code)
  Отправить код авторизации через Telegram Bot API.
- bool verify_auth_code (const std::string &chat_id, const std::string &code)
  Проверить введённый пользователем код авторизации.
- void ensure_bot_token ()
  Убедиться, что глобальный токен бота загружен.

4.11.1 Detailed Description

Интерфейс для Telegram-аутентификации: генерация, отправка и проверка кодов.

Описание:

- Использует Telegram Bot API для отправки одноразовых кодов авторизации.
- Хранит сгенерированные коды в глобальной карте auth_codes.

Definition in file telegram_auth.h.

4.11.2 Function Documentation

4.11.2.1 ensure_bot_token()

```
void ensure_bot_token ( )
```

Убедиться, что глобальный токен бота загружен.

При первом вызове пытается считать токен из файла SERVER_SETTINGS/BOT_TOKEN.txt. Если файл отсутствует или пуст, создаёт его и выводит сообщение об ошибке. Завершает программу при отсутствии токена.

Definition at line 30 of file telegram_auth.cpp.

```
00030                       {
00031     namespace fs = std::filesystem;
00032     fs::path dir = "SERVER_SETTINGS";
00033     fs::path file = dir / "BOT_TOKEN.txt";
00034
00035     if (!fs::exists(dir))
00036         fs::create_directories(dir);
00037
00038     if (fs::exists(file)) {
00039         std::ifstream in(file);
00040         std::getline(in, BOT_TOKEN);
00041     } else {
00042         std::ofstream out(file);
00043     }
00044
00045     if (BOT_TOKEN.empty()) {
00046         std::cerr « "[TelegramAuth] Файл " « file
00047                 « " не содержит токен.\n"
00048                   "Добавьте токен в первую строку и перезапустите сервер.\n";
00049         exit(1);
00050     }
00051 }
```

4.11.2.2 generate_auth_code()

```
std::string generate_auth_code ( )
```

Сгенерировать случайный шестизначный код для авторизации.

Код состоит из цифр [0-9] и всегда имеет длину 6 символов.

Returns

Сгенерированный код (например, "042517").

Definition at line 17 of file telegram_auth.cpp.

```
00017                           {
00018     static bool seeded = false;
00019     if (!seeded) {
00020         std::srand(static_cast<unsigned int>(std::time(nullptr)));
00021         seeded = true;
00022     }
00023
00024     std::string digits = "0123456789", code;
00025     for (int i = 0; i < 6; ++i)
00026         code += digits[std::rand() % 10];
00027     return code;
00028 }
```

4.11.2.3 send_telegram_code()

```
bool send_telegram_code (
                const std::string & chat_id,
                const std::string & code )
```

Отправить код авторизации через Telegram Bot API.

Формирует и выполняет HTTP-запрос для отправки сообщения с одноразовым кодом.

Parameters

chat↩ _id	Идентификатор Telegram-чата получателя.
code	Шестизначный код, который будет отправлен.

Returns

true, если сообщение успешно отправлено и код сохранён; false в случае ошибки при вызове curl или неверного ответа API.

Definition at line 53 of file telegram_auth.cpp.

```
00053                                                    {
00054     cpr::Response response = cpr::Post(
00055         cpr::Url{"https://api.telegram.org/bot" + BOT_TOKEN + "/sendMessage"},
00056         cpr::Payload{{"chat_id", chat_id}, {"text", "Your code is: " + code}}
00057     );
00058     if (response.text.find("\"ok\":true") != std::string::npos) {
00059         auth_codes[chat_id] = code;
00060         return true;
00061     }
00062     return false;
00063 }
```

4.11.2.4 verify_auth_code()

```
bool verify_auth_code (
                const std::string & chat_id,
                const std::string & code )
```

Проверить введённый пользователем код авторизации.

Сравнивает переданный code с сохранённым в auth_codes для данного chat_id.

Parameters

chat↩ _id	Идентификатор Telegram-чата, для которого код генерировался.
code	Код, введённый пользователем.

Returns

true, если код совпадает с ранее сгенерированным; иначе false.

Definition at line 65 of file telegram_auth.cpp.

```
00065                                                    {
00066     return auth_codes.count(chat_id) && auth_codes[chat_id] == code;
00067 }
```

4.12 telegram_auth.h

Go to the documentation of this file.

```
00001 /**
00002  * @file telegram_auth.h
00003  * @brief Интерфейс для Telegram-аутентификации: генерация, отправка и проверка кодов.
```

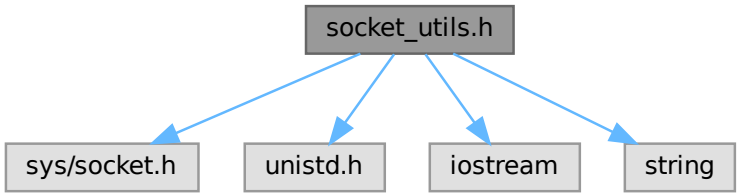
```
00004  *
00005  * Описание:
00006  * - Использует Telegram Bot API для отправки одноразовых кодов авторизации.
00007  * - Хранит сгенерированные коды в глобальной карте auth_codes.
00008  */
00009
00010 #ifndef TELEGRAM_AUTH_H
00011 #define TELEGRAM_AUTH_H
00012
00013 #include <string>
00014
00015 /**
00016  * @brief Сгенерировать случайный шестизначный код для авторизации.
00017  *
00018  * Код состоит из цифр [0-9] и всегда имеет длину 6 символов.
00019  *
00020  * @return Сгенерированный код (например, "042517").
00021  */
00022 std::string generate_auth_code();
00023
00024 /**
00025  * @brief Отправить код авторизации через Telegram Bot API.
00026  *
00027  * Формирует и выполняет HTTP-запрос для отправки сообщения с одноразовым кодом.
00028  *
00029  * @param chat_id Идентификатор Telegram-чата получателя.
00030  * @param code    Шестизначный код, который будет отправлен.
00031  * @return true, если сообщение успешно отправлено и код сохранён;
00032  *         false в случае ошибки при вызове curl или неверного ответа API.
00033  */
00034 bool send_telegram_code(const std::string& chat_id, const std::string& code);
00035
00036 /**
00037  * @brief Проверить введённый пользователем код авторизации.
00038  *
00039  * Сравнивает переданный @p code с сохранённым в auth_codes для данного @p chat_id.
00040  *
00041  * @param chat_id Идентификатор Telegram-чата, для которого код генерировался.
00042  * @param code    Код, введённый пользователем.
00043  * @return true, если код совпадает с ранее сгенерированным; иначе false.
00044  */
00045 bool verify_auth_code(const std::string& chat_id, const std::string& code);
00046
00047 /**
00048  * @brief Убедиться, что глобальный токен бота загружен.
00049  *
00050  * При первом вызове пытается считать токен из файла SERVER_SETTINGS/BOT_TOKEN.txt.
00051  * Если файл отсутствует или пуст, создаёт его и выводит сообщение об ошибке.
00052  * Завершает программу при отсутствии токена.
00053  */
00054 void ensure_bot_token();
00055
00056 #endif  // TELEGRAM_AUTH_H
```

4.13 socket_utils.h File Reference

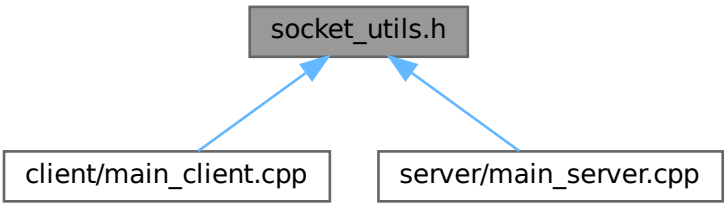
Обёртки функций отправки и приёма данных по TCP-сокетам.

#include <sys/socket.h>
#include <unistd.h>
#include <iostream>
#include <string>

Include dependency graph for socket_utils.h:



This graph shows which files directly or indirectly include this file:



Functions

- bool send_all (int fd, const std::string &msg)

  Отправить всю строку целиком по TCP-сокету.
- bool send_packet (int fd, std::string message)

  Отправить пакет текстовых данных через сокет.
- bool send_line (int fd, std::string message)

  Отправить одну строку через сокет.
- bool recv_line (int fd, std::string &out)

  Прочитать одну строку из сокета до символа новой строки.

4.13.1 Detailed Description

Обёртки функций отправки и приёма данных по TCP-сокетам.

Содержит inline-функции:

- send_all: отправить весь буфер данных;
- send_packet: отправить пакет строки с маркером конца сообщения "*ENDM*";

- send_line: отправить одну строку с терминатором '
';
- recv_line: получить одну строку до символа '
'.

Definition in file socket_utils.h.

4.13.2 Function Documentation

4.13.2.1 recv_line()

```
bool recv_line (
                int fd,
                std::string & out )   [inline]
```

Прочитать одну строку из сокета до символа новой строки.

Читает по одному символу через ::recv() и сохраняет их в out до встречи '
'. Символ '
' не включается.

Parameters

fd	Дескриптор сокета.
out	Переменная для сохранения прочитанной строки.

Returns

true если строка успешно прочитана, false при закрытии соединения или ошибке.

Definition at line 95 of file socket_utils.h.

```
00095                                       {
00096     out.clear();
00097     char ch{};
00098     while (true) {
00099         ssize_t n = ::recv(fd, &ch, 1, 0);
00100         if (n <= 0)
00101             return false;
00102         if (ch == '\n')
00103             break;
00104         out.push_back(ch);
00105     }
00106     return true;
00107 }
```

4.13.2.2 send_all()

```
bool send_all (
                int fd,
                const std::string & msg )   [inline]
```

Отправить всю строку целиком по TCP-сокету.

Функция многократно вызывает системный ::send(), пока не будет передан каждый байт строки msg. Рассчитана на блокирующий сокет — ::send() внутри может подождать, когда освободится буфер ядра.

Parameters

fd	Дескриптор открытого TCP-сокета.
msg	Строка, которую нужно передать (без копирования — используется её внутренний буфер).

Returns

true Если переданы все символы строки.

false Если ::send() вернул 0 (соединение закрыто) или < 0 (критическая ошибка).

Definition at line 37 of file socket_utils.h.

```
00037                                       { //Без const ссылка на временный std::string запрещена стандартом.
00038     size_t sent = 0;
00039     while (sent < msg.size()) {
00040         ssize_t n = ::send(fd,
00041                            msg.c_str() + sent,       // адрес нужного байта
00042                            msg.size()  - sent,
00043                            0);
00044         if (n <= 0)                 // ошибка или разрыв
00045             return false;
00046         sent += static_cast<size_t>(n);
00047     }
00048     return true;
00049 }
```

4.13.2.3 send_line()

```
bool send_line (
                int fd,
                std::string message )   [inline]
```

Отправить одну строку через сокет.

Гарантирует наличие символа новой строки '
' в конце и отправляет через send_all().

Parameters

fd	Дескриптор сокета.
sv	Строка для отправки.

Returns

true если строка успешно отправлена, false при ошибке.

Definition at line 78 of file socket_utils.h.

```
00078                                       {
00079     if (message.empty() || message.back() != '\n') {
00080         message.push_back('\n');
00081     }
00082     return send_all(fd, message);
00083 }
```

4.13.2.4 send_packet()

```
bool send_packet (
                int fd,
                std::string message )   [inline]
```

Отправить пакет текстовых данных через сокет.

Добавляет '
' в конец сообщения, если его нет, затем добавляет маркер "*ENDM*\n" и отправляет через send_all().

Parameters

fd	Дескриптор сокета.
data	Текст данных для отправки.

Returns

true если пакет успешно отправлен, false при ошибке.

Definition at line 61 of file socket_utils.h.

```
00061                                                {
00062     if (message.empty() || message.back() != '\n')
00063         message.push_back('\n');
00064     message += "*ENDM*\n";
00065     return send_all(fd, message);
00066 }
```

4.14 socket_utils.h

Go to the documentation of this file.

```
00001 /**
00002  * @file socket_utils.h
00003  * @brief Обёртки функций отправки и приёма данных по TCP-сокетам.
00004  *
00005  * Содержит inline-функции:
00006  *  - send_all: отправить весь буфер данных;
00007  *  - send_packet: отправить пакет строки с маркером конца сообщения "*ENDM*";
00008  *  - send_line: отправить одну строку с терминатором '\n';
00009  *  - recv_line: получить одну строку до символа '\n'.
00010  */
00011
00012 #ifndef SOCKET_UTILS_H
00013 #define SOCKET_UTILS_H
00014
00015 #include <sys/socket.h>
00016 #include <unistd.h>
00017
00018 #include <iostream>
00019 #include <string>
00020
00021 /**
00022  * @brief  Отправить всю строку целиком по TCP-сокету.
00023  *
00024  * Функция многократно вызывает системный ::send(), пока не
00025  * будет передан каждый байт строки @p msg.  Рассчитана на
00026  * **блокирующий** сокет — ::send() внутри может подождать,
00027  * когда освободится буфер ядра.
00028  *
00029  * @param fd   Дескриптор открытого TCP-сокета.
00030  * @param msg  Строка, которую нужно передать (без копирования —
00031  *             используется её внутренний буфер).
00032  *
00033  * @return true   Если переданы все символы строки.
00034  * @return false  Если ::send() вернул 0 (соединение закрыто)
00035  *                или < 0 (критическая ошибка).
00036  */
00037 inline bool send_all(int fd, const std::string& msg) { //Без const ссылка на временный std::string запрещена стандартом.
00038     size_t sent = 0;
00039     while (sent < msg.size()) {
00040         ssize_t n = ::send(fd,
00041                            msg.c_str() + sent,       // адрес нужного байта
00042                            msg.size()  - sent,
00043                            0);
00044         if (n <= 0)                 // ошибка или разрыв
```

```
00045             return false;
00046         sent += static_cast<size_t>(n);
00047     }
00048     return true;
00049 }
00050
00051 /**
00052  * @brief Отправить пакет текстовых данных через сокет.
00053  *
00054  * Добавляет '\n' в конец сообщения, если его нет,
00055  * затем добавляет маркер "*ENDM*\n" и отправляет через send_all().
00056  *
00057  * @param fd Дескриптор сокета.
00058  * @param data Текст данных для отправки.
00059  * @return true если пакет успешно отправлен, false при ошибке.
00060  */
00061 inline bool send_packet(int fd, std::string message) {
00062     if (message.empty() || message.back() != '\n')
00063         message.push_back('\n');
00064     message += "*ENDM*\n";
00065     return send_all(fd, message);
00066 }
00067
00068 /**
00069  * @brief Отправить одну строку через сокет.
00070  *
00071  * Гарантирует наличие символа новой строки '\n' в конце
00072  * и отправляет через send_all().
00073  *
00074  * @param fd Дескриптор сокета.
00075  * @param sv Строка для отправки.
00076  * @return true если строка успешно отправлена, false при ошибке.
00077  */
00078 inline bool send_line(int fd, std::string message) {
00079     if (message.empty() || message.back() != '\n') {
00080         message.push_back('\n');
00081     }
00082     return send_all(fd, message);
00083 }
00084
00085 /**
00086  * @brief Прочитать одну строку из сокета до символа новой строки.
00087  *
00088  * Читает по одному символу через ::recv() и сохраняет
00089  * их в @p out до встречи '\n'. Символ '\n' не включается.
00090  *
00091  * @param fd Дескриптор сокета.
00092  * @param out Переменная для сохранения прочитанной строки.
00093  * @return true если строка успешно прочитана, false при закрытии соединения или ошибке.
00094  */
00095 inline bool recv_line(int fd, std::string& out) {
00096     out.clear();
00097     char ch{};
00098     while (true) {
00099         ssize_t n = ::recv(fd, &ch, 1, 0);
00100         if (n <= 0)
00101             return false;
00102         if (ch == '\n')
00103             break;
00104         out.push_back(ch);
00105     }
00106     return true;
00107 }
00108
00109 #endif  // SOCKET_UTILS_H
```

Предметный указатель